Robustly Safe Compilation, an Efficient Form of Secure Compilation

MARCO PATRIGNANI, Stanford University, USA and CISPA Helmholz Center for Information Security, Germany
DEEPAK GARG, Max Planck Institute for Software Systems, Germany

Security-preserving compilers generate compiled code that withstands target-level attacks such as alteration of control flow, data leaks or memory corruption. Many existing security-preserving compilers are proven to be fully abstract, meaning that they reflect and preserve observational equivalence. Fully abstract compilation is strong and useful but, in certain cases, comes at the cost of requiring expensive runtime constructs in compiled code. These constructs may have no relevance for security, but are needed to accommodate differences between the source and target languages that fully abstract compilation necessarily needs.

As an alternative to fully abstract compilation, this paper explores a different criterion for secure compilation called robustly safe compilation or *RSC*. Briefly, this criterion means that the compiled code preserves relevant safety properties of the source program against all adversarial contexts interacting with the compiled program. We show that *RSC* can be proved more easily than fully abstract compilation and also often results in more efficient code. We also present two different proof techniques for establishing that a compiler attains *RSC* and, to illustrate them, develop three illustrative robustly-safe compilers that rely on different target-level protection mechanisms. We then proceed to turn one of our compilers into a fully abstract one and through this example argue that proving *RSC* can be simpler than proving fully abstraction.

To better explain and clarify notions, this paper uses syntax highlighting in a way that colourblind and black-&-white readers can benefit from [58]. For a better experience, please print or view this paper in colour.[1]

CCS Concepts: • **Security and privacy → Formal methods and theory of security**; • **Software and its engineering → Compilers**; *General programming languages.*

Additional Key Words and Phrases: secure compilation, robust safety, robustly-safe compilation, fully abstract compilation, formal languages, programming languages

ACM Reference Format:
Marco Patrignani and Deepak Garg. 2020. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2020), 42 pages. https://doi.org/10.1145/3436809

[1]Specifically, in this paper we use a blue, sans-serif font for elements of the source language, an **orange**, **bold** font for elements of the **first two target** languages and a *pink italics* font for elements of the *third target* language. Elements common to all languages are typeset in a *black*, *italic* font (to avoid repeating similar definitions twice), thus, C is a source-level component, **C** and *C* are target-level components and *C* is generic notation for either a source-level or a target-level component.

Authors' addresses: Marco Patrignani, Computer Science, Stanford University, Stanford, USA, , CISPA Helmholz Center for Information Security , Saarbrücken, Germany, mp@cs.stanford.edu; Deepak Garg, Max Planck Institute for Software Systems, Saarbrücken, Germany, dg@mpi-sws.org.

0164-0925/2020/1-ART1
https://doi.org/10.1145/3436809

1 INTRODUCTION

Low-level adversaries, such as those written in C or assembly can attack co-linked code written in a high-level language in ways that may not be feasible in the high-level language itself. For example, such an adversary may manipulate or hijack control flow, cause buffer overflows, or directly access private memory, all in contravention to the abstractions of the high-level language. Specific countermeasures such as Control Flow Integrity [3] or Code Pointer Integrity [44] have been devised to address some of these attacks *individually*. An alternative approach is to devise a *security-preserving compiler*, which seeks to defend against entire *classes* of such attacks. Security-preserving compilers often achieve security by relying on different protection mechanisms, e.g., cryptographic primitives [4, 5, 24, 28], types [11, 12], address space layout randomisation [6, 40], protected module architectures [10, 59, 61, 63] (also know as enclaves [49]), tagged architectures [7, 42], etc. Once designed, the question researchers face is how to formalise that such a compiler is indeed secure, and how to prove this. Basically, we want a criterion that specifies secure compilation. A widely-used criterion for compiler security is fully abstract compilation (*FAC*) [1, 38, 56], which has been shown to preserve many interesting security properties like confidentiality, integrity, invariant definitions, well-bracketed control flow and hiding of local state [10, 40, 59, 60].

Informally, a compiler is fully abstract if it preserves and reflects observational equivalence of source-level components (i.e., partial programs) in their compiled counterparts. Most existing work instantiates observational equivalence with contextual equivalence: co-divergence of two components in any larger context they interact with. Fully abstract compilation is a very strong property, which preserves *all* source-level abstractions.

Unfortunately, preserving *all* source-level abstractions also has downsides. In fact, while *FAC* preserves many relevant security properties, it also preserves a plethora of other non-security ones, and the latter may force inefficient checks in the compiled code. For example, when the target is assembly, two observationally equivalent components must compile to code of the same size [10, 59], else full abstraction is trivially violated. This requirement is security-irrelevant in most cases. Additionally, *FAC* is not well-suited for source languages with undefined behaviour (e.g., `C` and `LLVM`) [42] and, if used naïvely, it can fail to preserve even simple safety properties [64] (though, fortunately, no *existing* work falls prey to this naïvety).

Motivated by this, recent work started investigating alternative secure compilation criteria that overcome these limitations. These security-focussed criteria take the form of preservation of hyperproperties or classes of hyperproperties, such as hypersafety properties or safety properties [9, 36]. This paper investigates one of these criteria, namely, *Robustly Safe Compilation* (*RSC*) which has clear security guarantees and, as we show, can often be attained more efficiently than FAC.

Informally, a compiler attains *RSC* if it is correct and it preserves *robust safety* of source components in the target components it produces. Robust safety is an important security notion that has been widely adopted to formalise security, e.g., of communication protocols [15, 19, 37]. Before explaining *RSC*, we explain robust safety as a language property.

Robust Safety as a Language Property. Informally, a program property is a safety property if it encodes that “bad” sequences of events do not happen when the program executes [14, 69]. A program is *robustly safe* if it has relevant (specified) safety properties *despite* active attacks from adversaries [15, 68, 76]. As the name suggests, robust safety relies on the notions of safety and robustness which we now explain.

Safety. As mentioned, safety asserts that “no bad sequence of events happens”, so we can specify a safety property by the set of *finite observations* which characterise all bad sequences of events. A whole program has a safety property if its behaviours exclude these bad observations. Many security properties can be encoded as safety, including integrity, weak secrecy and functional correctness.

Example 1.1 (Integrity). Integrity ensures that an attacker does not tamper with invariants on state. For example, consider the function `charge_account( amount )` in the snippet below, which deducts `amount` from an account as part of an electronic card payment. A card PIN is required if `amount` is larger than 10 euros. So the function checks whether `amount` > 10, requests the PIN if this is the case, and then changes the account balance. We expect this function to have a safety (integrity) property on the account balance: a reduction of more than 10 euros to the account balance must be preceded by a call to `request_pin()`. Here, the relevant observation is a trace (sequence) of account balances and calls to `request_pin()`. Bad observations for this safety property are those where an account balance is at least 10 euros less than the previous one, without a call to `request_pin()` in between. Note that this function seems to have this safety property, but it may not have the safety property *robustly*: a target-level adversary may transfer control directly to the "else" branch of the check `amount` > 10 after setting `amount` to more than 10, to violate the safety property.

```
function charge_account( amount : Int ){
 if amount > 10 { request_pin(); }
 charge_account(amount);
 return;
}
```

⊡

Example 1.2 (Weak Secrecy). Weak secrecy asserts that a program secret never flows *explicitly* to the attacker. For example, consider code that manages `network_h`, a handler (socket descriptor) for a sensitive network interface. This code does not expose `network_h` directly to external code but it provides an API to use it. This API makes some security checks internally. If the handler is directly accessible to outer code, then it can be misused in insecure ways (since the security checks may not be made). If the code has weak secrecy with respect to `network_h` then we know that the handler is never passed to an attacker. In this case we can define bad observations as those where `network_h` is passed to external code (e.g., as a parameter, as a return value on or on the heap). ⊡

Example 1.3 (Partial Correctness). Program correctness can also be formalised as a safety property. Consider a program that computes the `n`th Fibonacci number. The program reads `n` from an input source and writes its output to an output source. Correctness of this program is a safety property. Observations here are pairs of an input (read by the program) and the corresponding output (produced by the program) so, for example, outputting `13` is only allowed if `7` were passed as input. A bad observation is one where the input is `n` (for some `n`) but the output is different from the `n`th Fibonacci number, e.g., input `4` and output `5` as well as input `3` and output `6` are bad observations. ⊡

These examples not only illustrate the expressiveness of safety properties, but also show that safety properties as we capture here are quite *coarse-grained*, since they are only concerned with (sequences of) relevant events like calls to specific functions, changes to specific heap variables, inputs, and outputs. In the model of observable events that we use, we can see that safety properties do not specify or constrain how the program computes between these events, leaving the programmer and the compiler considerable flexibility in optimizations. This gives us confidence in the model of events we choose. However, safety properties are not a panacea for security, and there are security properties that are not safety. For example, noninterference [80, 82], the standard information flow property, is not safety. Nonetheless, many interesting security properties are safety. In fact, many non-safety properties including noninterference can be conservatively approximated as safety properties [22]. Hence, safety properties are a meaningful goal to pursue for secure compilation.

Robustness. We often want to reason about properties of a component of interest that hold irrespective of any other components the component interacts with. These other components may be the libraries the component is linked against, or the language runtime. Often, these surrounding components are modelled as the *program context* whose hole the component of interest fills. When the component of interest links to a context, we have a whole program that can run. A property holds *robustly* for a component if it holds in *any* context that the component of interest can be linked to.

From a security perspective, the context represents the attacker in the threat model we consider. The implications of this fact are that the attacker's power is limited to what can be expressed by the language semantics. Concretely, all of the attackers we consider have no control over the code section of the component and they cannot tamper with the protection mechanisms the compiler uses.

Robust Safety Preservation as a Compiler Property. A compiler attains robustly safe compilation or *RSC* if it maps any source component that has a safety property *robustly* to a compiled component that has the *same* safety property robustly. Thus, safety has to hold robustly in the target language, which often does not have the powerful abstractions (e.g., typing) that the source language has. Hence, the compiler must insert enough defensive runtime checks into the compiled code to prevent the more powerful *target* contexts from launching attacks (violations of safety properties) that source contexts could not launch. This is unlike correct compilation, which either considers only those target contexts that behave like source contexts [43, 53, 73] or considers only whole programs [46].

As mentioned, safety properties are usually quite coarse-grained. This means that *RSC* still allows the compiler to optimise code internally, as long as the sequence of observable events is not affected. For example, when compiling the `fibonacci` function of Example 1.3, the compiler can do any internal optimisation such as caching intermediate results, as long as the end result is correct. Crucially, however, cached results must be protected from tampering by a (target-level) attacker, else the output can be incorrect, breaking *RSC*.

A *RSC*-attaining compiler focuses only on preserving security (as captured by robust safety) instead of contextual equivalence (typically captured by full abstraction). So, such a compiler can produce code that is more efficient than code compiled with a fully abstract compiler as it does not have to preserve *all* source abstractions (we illustrate this later).

Finally, robust safety scales naturally to thread-based concurrency [2, 37, 62]. Thus *RSC* also scales naturally to thread-based concurrency (we demonstrate this too). This is unlike *FAC*, where thread-based concurrency can introduce additional undesired observations that also need to be preserved.

RSC is a very recently proposed criterion for security-preserving compilers. Recent work [8, 9, 36] defines *RSC* abstractly in terms of preservation of program behaviours, but the development is limited to the definition only. Other recent work [7] defines a form of *RSC* for source languages with undefined behaviour and where attackers are components that become compromised as execution progresses. Our goal in this paper is to examine how *RSC* can be realized and established, and to show that in certain cases it leads to compiled code that is more efficient than what *FAC* leads to. To this end, we consider a specific setting where observations are values in specific (sensitive) heap locations at cross-component calls. We define robust safety and *RSC* for this specific setting (Section 2). Unlike previous work [9, 15, 36] which assumed that the domain of traces (behaviours) is the same in the source and target languages, our *RSC* definition allows for different trace domains in the source and target languages, as long as they can be suitably related. This relation is analogous to that found in recent work [8] that studied the necessary properties of trace relations in order to preserve security through compilation. The second contribution of our paper is two proof techniques to establish *RSC*.

- The first technique is an adaption of trace-based backtranslation, an existing technique for proving *FAC* [7, 10, 63]. To illustrate this technique, we build a compiler from an untyped source language to an untyped target language with support for fine-grained memory protection via so-called capabilities [25, 81] (Section 3). Here, we guarantee that if a source program is robustly safe, then so is its compilation.
- The second proof technique shows that if source programs are *verified* for robust safety, then one can simplify the proof of *RSC* so that no backtranslation is needed. In this case, we develop a compiler from a *typed* source language where the types already enforce robust safety, to a target language similar to that of the first compiler (Section 4). In this instance, both languages also support shared-memory concurrency. Here, we guarantee that all compiled target programs are robustly safe.

To argue that *RSC* is general and is not limited to compilation targets based on capabilities, we also develop a third compiler.

- This compiler starts from the same source language as our second compiler but targets an untyped concurrent language with support for *coarse-grained memory isolation*, modelling recent hardware extensions such as Intel's SGX [49] (Section 5).

The final contribution of this paper is a comparison between *RSC* and *FAC* (Section 6). For this,

- We first introduce *FAC* and discuss its advantages and limitations.
- Then, we present a series of code examples that describe different ways in which a fully abstract compiler introduces inefficiencies in compiled code in order to attain *FAC*. We then sketch a fourth compiler by turning the first one into a fully abstract one and show how the changes introduced to attain *FAC* make compiled code inefficient.
- Finally, we argue that this compiler attains *FAC* and highlight how the proof is significantly more complex than before.

Finally, the paper discusses related work (Section 7) and concludes (Section 8).

This paper supersedes and extends the work of Patrignani and Garg [66] by providing full details of the languages and compilers formalisations. Additionally, it describes how the *RSC* theory scales to different protection mechanisms (Section 5) and it presents in much more detail the comparison with *FAC*. For the sake of brevity and clarity, we limit proofs to sketches, the interested reader will find full proofs and additional lemmas in the companion technical report [65].

2 ROBUSTLY SAFE COMPILATION

This section first discusses robust safety as a language (not a compiler) property (Section 2.1) and then presents *RSC* as a compiler property along with an informal discussion of techniques to prove it (Section 2.2).

2.1 Safety and Robust Safety

To explain robust safety, we first describe a general *imperative* programming model that we use. Programmers write *components* on which they want to enforce safety properties robustly. A component is a list of function definitions that can be linked with other components (the context) in order to obtain a runnable whole program (functions in "other" components are like `extern` functions in C). Additionally, every component declares a set of "sensitive" locations that contain all the data that is safety-relevant. For instance, in Example 1.1 this set may contain the account balance and in Example 1.3 it may contain the I/O buffers. We explain the relevance of this set after we define safety properties.

We want safety properties to specify that a component never executes a "bad" sequence of events. For this, we first need to fix a notion of events. We have several choices here, e.g., our events could be inputs and outputs, all syscalls, all changes to the heap (as in CompCert [47]), etc. Here, we make a specific choice motivated by our interest in robustness: we define events as calls/returns that cross a component boundary, together with the state of the heap at that point. Consequently, our safety properties can constrain the contents of the heap at component boundaries. This choice of component boundaries as the point of observation is meaningful because, in our programming model, control transfers to/from an adversary happen only at component boundaries (more precisely, they happen at cross-component function call and returns). This allows the compiler complete flexibility in optimizing code within a component, while not reducing the ability of safety properties to constrain observations of the adversary. In turn, safety properties regarding these kinds of boundary-crossing events are the only one that our criterion can preserve through compilers upholding our criterion.

Concretely, a component behaviour is a *trace*, i.e., a sequence of *actions* recording component boundary interactions and, in particular, the heap at these points. *Actions*, the items on a trace, have the following grammar (notation-wise, we mainly indicate actions as α, though to further disambiguate when source and target actions are mentioned, we will also use the ω notation):

$$\textit{Actions}\ \alpha, \omega ::= \texttt{call}\ f\ v\ H? \mid \texttt{call}\ f\ v\ H! \mid \texttt{ret}\ H! \mid \texttt{ret}\ H?$$

These actions respectively capture call and callback to a function f with parameter v when the heap is H as well as return and returnback with a certain heap H. More precisely, a callback is a call from the component to the context, so it generates label $\texttt{call}\ f\ v\ H!$ while a returnback is a return from such a callback, i.e., the context returning to the component, and it generates the label $\texttt{ret}\ H?$. We use ? and ! decorations to indicate whether the control flow of the action goes from the context to the component (?) or from the component to the context (!). Well-formed traces have alternations of ? and ! decorated actions, starting with ? since execution starts in the context. For a sequence of actions $\overline{\alpha}$, $\texttt{relevant}(\overline{\alpha})$ is the list of heaps $\overline{H}$ mentioned in the actions of $\overline{\alpha}$. In the sequent, we separate list elements with $\cdot$, so $\overline{H} \cdot H$ indicates a non-empty list of heaps with at least one element (H).

Next, we need a representation of safety properties. Generally, properties are sets of traces, but safety properties specifically can be specified as automata (or monitors in the sequel) [69]. We choose this representation since monitors are less abstract than sets of traces and they are closer to enforcement mechanisms used for safety properties, e.g., runtime monitors. Briefly, a safety property is a monitor that transitions states in response to events of the program trace. At any point, the monitor may refuse to transition (it gets *stuck*), which encodes property violation. While a monitor can transition, the property has not been violated. Schneider [69] argues that all properties codable this way are safety properties and that all enforceable safety properties can be coded this way.

Formally, a monitor M in our setting consists of a set of abstract states $\{\sigma \cdots\}$, the transition relation $\rightsquigarrow$, an initial state σ_0, the set of heap locations that matter for the monitor, $\{l \cdots\}$, and the current state σ_c (we indicate a set of elements of class e as $\{e \cdots\}$). The transition relation $\rightsquigarrow$ is a set of triples of the form (σ_s, H, σ_f) consisting of a starting state σ_s, a final state σ_f and a heap H. The transition (σ_s, H, σ_f) is interpreted as "*state σ_s transitions to σ_f when the heap is H*". When determining the monitor transition in response to a program action, we restrict the program's heap to the location set $\{l \cdots\}$, i.e., to the set of locations the monitor cares about. This heap restriction is written $H\big|_{\{l\cdots\}}$. We assume determinism of the transition relation: for any σ_s and (restricted heap) H, there is at most one σ_f such that $(\sigma_s, H, \sigma_f) \in \rightsquigarrow$.

Given the behaviour of a program as a trace $\overline{\alpha}$ and a monitor M specifying a safety property, $M \vdash \overline{\alpha}$ denotes that the trace satisfies the safety property. Intuitively, to satisfy a safety property, the sequence of heaps in the actions of a trace, *restricted to the locations that the monitor cares about*, must never get the monitor stuck (Rule Valid trace). Every single restricted heap must allow the monitor to step according to its transition relation (Rule Monitor Step). Note that we overload the $\rightsquigarrow$ notation here to also denote an auxiliary relation, the *monitor small-step semantics* (Rule Monitor Step-base and Rule Monitor Step-ind).

$$\frac{M; \texttt{relevant}(\overline{\alpha}) \rightsquigarrow M'}{M \vdash \overline{\alpha}}\;\text{(Valid trace)} \qquad \frac{}{M; \varnothing \rightsquigarrow M}\;\text{(Monitor Step-base)} \qquad \frac{M; \overline{H} \rightsquigarrow M'' \qquad M''; H \rightsquigarrow M'}{M; \overline{H} \cdot H \rightsquigarrow M'}\;\text{(Monitor Step-ind)}$$

$$\frac{(\sigma_c, H\big|_{\{l\cdots\}}, \sigma_f) \in \rightsquigarrow}{(\{\sigma\cdots\}, \rightsquigarrow, \sigma_0, \{l\cdots\}, \sigma_c); H \rightsquigarrow (\{\sigma\cdots\}, \rightsquigarrow, \sigma_0, \{l\cdots\}, \sigma_f)}\;\text{(Monitor Step)}$$

With this setup in place, we can formalise safety, attackers and robust safety. In defining (robust) safety for a component, we only admit monitors (safety properties) whose $\{l\cdots\}$ agrees with the sensitive locations declared by the component. Making the set of safety-relevant locations explicit in the component and the monitor gives the compiler more flexibility by telling it precisely which locations need to be protected against target-level attacks (the compiler may choose to not protect the rest). At the same time, it allows for expressive modelling. For instance, in Example 1.3, the safety-relevant locations could be the I/O buffers from which the program performs inputs and outputs, and the safety property can constrain the input and output buffers at corresponding call and return actions involving the Fibonacci function. A whole program C is safe for a monitor M, written $M \vdash C : \mathit{safe}$, if the monitor accepts any trace the program generates from its initial state $(\Omega_0\,(C))$.

An attacker A is valid for a component C, written $C \vdash A : \mathit{atk}$, if A's free locations (denoted $\texttt{locs}(A)$) are disjoint from the locations that the component cares about (denoted $C.\texttt{locs}$). This is a basic sanity check: if we allow an attacker to mention heap locations that the component cares about, the attacker will be able to modify those locations, causing all but trivial safety properties to *not* hold robustly.

A component C is robustly safe with respect to monitor M, written $M \vdash C : \mathit{rs}$, if C composed with *any* attacker is safe with respect to M. As mentioned, for this setup to make sense, the monitor and the component must agree on the locations that are safety-relevant. This agreement is denoted $M \frown C$.

Definition 2.1 (Safety, attacker and robust safety).

$$M \vdash C : \mathit{safe} \overset{\text{def}}{=} \text{ if } \vdash C : \mathit{whole} \text{ then if } \Omega_0\,(C) \overset{\overline{\alpha}}{\Longrightarrow} \_ \text{ then } M \vdash \overline{\alpha}$$

$$C \vdash A : \mathit{atk} \overset{\text{def}}{=} C.\texttt{locs} = \{l\cdots\} \text{ and } \{l\cdots\} \cap \texttt{locs}(A) = \varnothing$$

$$M \vdash C : \mathit{rs} \overset{\text{def}}{=} \forall A.\text{ if } M \frown C \text{ and } C \vdash A : \mathit{atk} \text{ then } M \vdash A\,[C] : \mathit{safe}$$

2.2 Robustly Safe Compilation

Robustly-safe compilation ensures that robust safety properties *and their meanings* are preserved across compilation. But what does it means to preserve meanings across languages? If a source safety property says never write 3 to a location, and we compile to an assembly language by mapping numbers to binary, the corresponding target property should say **never write 0x11 to an address**.

In order to relate properties across languages, we assume a relation $\approx : v \times v$ between source and target values that is *total* in the first component, so it maps any source value v to a target value v: $\forall v. \exists v. v \approx v$. This value relation is used to define a relation between heaps: $H \approx H$, which intuitively holds when related locations point to related values. This is then used to define a relation between actions: $\alpha \approx \omega$, which holds when the two actions are the "same" modulo this relation, i.e., `call` $\cdots$? only relates to `call` $\cdots$? and the arguments of the action (values and heap) are related. Next, we require a relation $M \approx M$ between source and target monitors, which means that the source monitor M and the target monitor M enforce the same safety property, modulo the relation $\approx$ on actions (and thus on locations and values too) assumed above. The precise definition of this relation depends on the source and target languages; specific instances are shown in Sections 3.3.1 and 4.3.[2]

We denote a compiler from language S to language T by $[\![\cdot]\!]^{S}_{T}$. A compiler $[\![\cdot]\!]^{S}_{T}$ attains *RSC*, if it maps any component C that is robustly safe with respect to M to a component C that is robustly safe with respect to M, provided that $M \approx M$.

Definition 2.2 (Robustly Safe Compilation).

$$\vdash [\![\cdot]\!]^{S}_{T} : RSC \overset{\text{def}}{=} \forall C, M, M.\ \text{ if } M \vdash C : rs \text{ and } M \approx M \text{ then } M \vdash [\![C]\!]^{S}_{T} : rs$$

A consequence of the universal quantification over monitors here is that the compiler cannot be property-sensitive. A robustly-safe compiler preserves all robust safety properties, not just a specific one, e.g., it does not just enforce that `fibonacci` is correct. This seemingly strong goal is sensible as compiler writers will likely not know what safety properties individual programmers will want to preserve.

Remark #1: Safety Through Assertions. Some readers may wonder why we do not follow existing work and specify safety as "programmer-written assertions never fail" [34, 37, 48, 76]. Unfortunately, this approach does not yield a meaningful criterion for specifying a compiler, since assertions in the compiled program (if any) are generated by the compiler itself. Thus a compiler could just erase all assertions and the compiled code it generates would be trivially (robustly) safe – no assertion can fail if there are no assertions in the first place!

Remark #2: Compiling Monitors. In our development, we assume that a source and a target monitor are related and do not actually compile a source monitor to obtain a related target monitor. While such compilation is feasible, it is at odds with our view of monitors as *specifications* of safety properties. Compiling monitors and, in particular, compiling monitors with the same compiler that we want to prove security of, leads to a circularity—we must understand the compiler to understand the target safety property, which, in turn, acts as the specification for the compiler! Consequently, we choose not to compile monitors and talk only of an abstract, compiler-independent relation between source and target monitors.

2.2.1 Proving RSC. Proving that a compiler attains *RSC* can be done either by proving that a compiler satisfies Definition 2.2 or by proving something *equivalent*. To this end, Definition 2.3 below presents an alternative, equivalent formulation of *RSC*. We call this characterisation *property-free* as it does not mention monitors explicitly (it mentions the `relevant(·)` function for reasons we explain below).

[2]Accounting for the difference in the representation of safety properties sets us apart from recent work [9, 36], which assumes that the source and target languages have the same trace alphabet. The latter works only in some settings.

Definition 2.3 (Property-Free RSC).

$$\vdash [\![\cdot]\!]^{S}_{T} : \textit{PF-RSC} \overset{\text{def}}{=} \forall C, A, \overline{\omega}.$$

$$\text{if } [\![C]\!]^{S}_{T} \vdash A : \mathbf{atk} \text{ and } \vdash A\left[[\![C]\!]^{S}_{T}\right] : \mathbf{whole} \text{ and } \Omega_0\left(A\left[[\![C]\!]^{S}_{T}\right]\right) \xRightarrow{\overline{\omega}} \_$$

$$\text{then } \exists A, \overline{\alpha}.\ C \vdash A : \mathrm{atk} \text{ and } \vdash A\,[C] : \mathrm{whole} \text{ and } {}^{\cdot}_{0}\,(A\,[C]) \xRightarrow{\overline{\alpha}} \_$$

$$\text{and } \texttt{relevant}(\overline{\alpha}) \approx \texttt{relevant}(\overline{\omega})$$

PF-RSC states that if the compiled code produces a behaviour in a target context, then the source code also produces a related behaviour in *some* source context. In other words, target contexts cannot induce more (bad) behaviours in the compiled code than source contexts can in the source code.

PF-RSC and *RSC* should, in general, be equivalent (Proposition 2.4).

Proposition 2.4 (*PF-RSC* and *RSC* are equivalent).

$$\forall [\![\cdot]\!]^{S}_{T}, \vdash [\![\cdot]\!]^{S}_{T} : \textit{PF-RSC} \iff \vdash [\![\cdot]\!]^{S}_{T} : \textit{RSC}$$

As mentioned in Section 1, a property is safety if it asserts that "no bad sequence of events happens", so a safety property specifies the set of *bad prefixes* (i.e., finite traces) which characterise all bad sequences of events. As such, a safety property implies that programs do not have any trace prefix from the set of bad prefixes. Hence, *not* having a safety property robustly amounts to some context being able to induce a bad prefix. Consequently, preserving *all* robust safety properties (*RSC*) amounts to ensuring that all target prefixes can be generated (by some context) in the source too (*PF-RSC*). Formally, since Definition 2.2 relies on the monitor relation, we can prove Proposition 2.4 only after such a relation is finalised. We give such a monitor relation and proof in Section 3.3 (see Theorem 3.7). However, in general this result should hold for any cross-language monitor relation that correctly relates safety properties. If the proposition does not hold, then the relation does not capture how safety in one language is represented in the other.

Assuming Proposition 2.4, we can prove *PF-RSC* for a compiler in place of *RSC*. *PF-RSC* can be proved with a *backtranslation* technique. This technique has been often used to prove full abstraction [7, 9, 10, 36, 42, 54, 59, 60, 63] and it aims at building a source context starting from a target one. In fact *PF-RSC*, leads directly to a backtranslation-based proof technique since it can be rewritten (eliding irrelevant details) as:

$$\forall \overline{\omega} \text{ if } \exists A.\ \Omega_0\left(A\left[[\![C]\!]^{S}_{T}\right]\right) \xRightarrow{\overline{\omega}} \_$$

$$\text{then } \exists A, \overline{\alpha}.\ {}^{\cdot}_{0}\,(A\,[C]) \xRightarrow{\overline{\alpha}} \_ \text{ and } \texttt{relevant}(\overline{\alpha}) \approx \texttt{relevant}(\overline{\omega})$$

Essentially, given a target context A, a compiled program $[\![C]\!]^{S}_{T}$ and a target trace $\overline{\omega}$ that A causes $[\![C]\!]^{S}_{T}$ to have, we need to construct, or *backtranslate* to, a source context A that will cause the source program C to simulate $\overline{\omega}$. Such backtranslation based proofs can be quite difficult, depending on the features of the languages and the compiler. However, backtranslation for *RSC* (as we show in Section 3.3.1) is not as complex as backtranslation for *FAC* (Section 6.3).

A simpler proof strategy is also viable for *RSC* when we compile only those source programs that have been *verified* to be robustly safe (e.g., using a type system). The idea is this: from the verification of the source program, we can find an invariant which is always maintained by the target code, and which, in turn, implies the robust safety of the target code. For example, if the safety property is that values in the heap always have their expected types, then the invariant can simply be that values in the target heap are always related to the source ones (which have their expected types). This is tantamount to proving type preservation in the target in the presence of an active adversary. This is harder than standard type preservation (because of the active adversary) but is still much easier than backtranslation as there is no need to map target constructs to source contexts syntactically. We illustrate this proof technique in Section 4.

2.2.2 RSC Implies Compiler Correctness. As stated in Section 1, *RSC* implies (a form of) compiler correctness. While this may not be apparent from Definition 2.2, it is more apparent from its equivalent characterization in Definition 2.3. We elaborate this here.

Whether concerned with whole programs or partial programs, compiler correctness states that the behaviour of compiled programs *refines* the behaviour of source programs [20, 39, 43, 47, 53, 73]. So, if $\{\overline{\omega} \cdots\}$ and $\{\overline{\alpha} \cdots\}$ are the sets of compiled and source behaviours, then a compiler should force $\{\overline{\omega} \cdots\} \precapprox \{\overline{\alpha} \cdots\}$, where $\precapprox$ is the composition of $\subseteq$ and of the relation $\approx^{-1}$.

If we consider a source component C that is whole, then it can only link against empty contexts, both in the source and in the target. Hence, in this special case, *PF-RSC* simplifies to standard refinement of traces, i.e., whole program compiler correctness. Hence, assuming that the correctness criterion for a compiler is concerned with the same observations as safety properties (values in safety-relevant heap locations at component crossings in our illustrative setting), *PF-RSC* implies whole program compiler correctness.

However, *PF-RSC* (or, equivalently, *RSC*) does not imply, nor is implied by, any form of *compositional compiler correctness* (CCC) [43, 53, 73]. CCC requires that the behaviours produced by a compiled component linked against a target context that is related (in behaviour) to a source context can also be produced by the source component linked against the *related* source context. In contrast, *PF-RSC* allows picking *any* source context to simulate the behaviours. Hence, *PF-RSC* does not imply CCC. On the other hand, *PF-RSC* universally quantifies over all target contexts, while CCC only quantifies over target contexts related to a source context, so CCC does not imply *PF-RSC* either. Hence, compositional compiler correctness, if desirable, must be imposed in addition to *PF-RSC*.

We could remedy this and generalise our criterion even more by adding an additional parameter, a relation between source and target contexts that binds the quantified target and source contexts. Our criterion chooses the weakest of these relations, where all source contexts are related to all target ones, in order to not impose any constraints on A thus making the attackers in our threat model as powerful as possible. Existing compositional compiler correctness criteria would instantiate this relation e.g., between a source context and its compilation [43, 73] or between a source context and something that behaves like its compilation [53]. As we focus on security, we choose not to pollute our definition with an additional parameter and leave this relation out.

Note that the lack of implications between *PF-RSC* and CCC is unsurprising: the two criteria capture two very different aspects of compilation: security (against all contexts) and compositional preservation of behaviour (against well-behaved contexts).

Remark. Compiler correctness composes ‘vertically’, that is, given two compilers (or, compiler passes) one from a source language to an intermediate one, and one from the intermediate to a target language, the compiler resulting of the composition of the two passes is still correct. Like compiler correctness, *PF-RSC* also composes vertically, i.e., if several compiler passes are all *PF-RSC*, then the compiler resulting of the composition of those passes is also *PF-RSC*. Studying how *PF-RSC* compiler passes interact with other passes (e.g., compiler passes that may be *FAC* or any other criterion from [9]) is an open research question.

3 *RSC* VIA TRACE-BASED BACKTRANSLATION

This section illustrates how to prove that a compiler attains *RSC* by means of a trace-based back-translation technique [7, 59, 63]. To present such a proof, we first introduce our source language $\mathsf{L^U}$, an untyped, first-order imperative language with abstract references and hidden local state (Section 3.1). Then, we present our target language $\mathbf{L^P}$, an untyped imperative target language with a concrete heap, whose locations are natural numbers that the context can compute. $\mathbf{L^P}$ provides hidden local state via a fine-grained capability mechanism on heap accesses (Section 3.2). Finally, we present the compiler $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ and prove that it attains *RSC* (Section 3.3) by means of a trace-based backtranslation. The section concludes with an example detailing why *RSC* preserves security (Example 3.8).

To avoid focussing on mundane details, we deliberately use source and target languages that are fairly similar. However, they differ substantially in one key point: the heap model. This affords the target-level adversary attacks like guessing private locations and writing to them that do not obviously exist in the source (and makes our proofs nontrivial). We believe that (with due effort) the ideas here will generalize to languages with larger gaps and more features.

3.1 The Source Language $\mathsf{L^U}$

$$
\begin{array}{rlrl}
\textit{Components}\ \mathsf{C} ::= & \ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}} & \textit{Contexts}\ \mathsf{A} ::= & \mathsf{H}; \overline{\mathsf{F}}\ [\cdot] \\
\textit{Interfaces}\ \mathsf{I} ::= & \mathsf{f} & \textit{Functions}\ \mathsf{F} ::= & \mathsf{f(x)} \mapsto \mathsf{s; return;} \\
\textit{Heaps}\ \mathsf{H} ::= & \varnothing \mid \mathsf{H}; \ell \mapsto \mathsf{v} & \textit{Values}\ \mathsf{v} ::= & \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{n} \in \mathbb{N} \mid \langle \mathsf{v}, \mathsf{v} \rangle \mid \ell
\end{array}
$$

$$
\begin{array}{rl}
\textit{Expressions}\ \mathsf{e} ::= & \mathsf{x} \mid \mathsf{v} \mid \mathsf{e} \oplus \mathsf{e} \mid \mathsf{e} \otimes \mathsf{e} \mid \langle \mathsf{e}, \mathsf{e} \rangle \mid \mathsf{e.1} \mid \mathsf{e.2} \mid \mathsf{!e} \\
\textit{Statements}\ \mathsf{s} ::= & \mathsf{skip} \mid \mathsf{s; s} \mid \mathsf{let\ x = e\ in\ s} \mid \mathsf{if\ e\ then\ s\ else\ s} \mid \mathsf{call\ f\ e} \mid \mathsf{let\ x = new\ e\ in\ s} \mid \mathsf{x := e} \\
\textit{Eval. Ctxs.}\ \mathsf{E} ::= & [\cdot] \mid \mathsf{n} \oplus \mathsf{E} \mid \mathsf{E} \oplus \mathsf{e} \mid \mathsf{n} \otimes \mathsf{E} \mid \mathsf{E} \otimes \mathsf{e} \mid \langle \mathsf{v}, \mathsf{E} \rangle \mid \langle \mathsf{E}, \mathsf{e} \rangle \mid \mathsf{E.1} \mid \mathsf{E.2} \mid \mathsf{!E}
\end{array}
$$

$$
\begin{array}{rlrl}
\textit{Mon. States}\ \sigma \in & \mathcal{S} & \textit{Monitors}\ \mathsf{M} ::= & (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_c) \\
\textit{Mon. Reds.}\ \rightsquigarrow ::= & \varnothing \mid \rightsquigarrow; (\mathsf{s}, \mathsf{H}, \mathsf{s}) & \textit{Prog. States}\ ^{\cdot} ::= & \mathsf{C}, \mathsf{H} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}} \\
\textit{Labels}\ \lambda ::= & \epsilon \mid \alpha & \textit{Actions}\ \alpha ::= & \mathsf{call\ f\ v\ H?} \mid \mathsf{call\ f\ v\ H!} \mid \mathsf{ret\ H!} \mid \mathsf{ret\ H?}
\end{array}
$$

Fig. 1. Syntax of $\mathsf{L^U}$. We indicate a list of elements $e_1, \cdots, e_n$ as $\overline{e}$.

$\mathsf{L^U}$ is an untyped imperative while language [55]. Its syntax is presented in Figure 1. Components C are triples of function definitions, interfaces and a special location written ℓ_{root}, which defines the locations that are monitored for safety, as explained below. We use a mnemonic 'dot' notation to access sub-parts of elements that are tuples (such as components), so when accessing the functions sub-part of a component C, we will write C.`funs`. Each function definition maps a function name and a formal argument to a body s. An interface is a list of functions that the component relies on the context to provide (similar to C's `extern` declarations). Attackers A (program contexts) are function definitions (and their heap) that represent untrusted code that a component interacts with. A function body is a statement. Statements s are rather standard but their treatment is not. For example, statements define local variables but these are substitute and not looked up in an environment as in while languages [79]. Additionally, statements manipulate the heap, do recursive function calls and branch conditionally. Statements use effect-free expressions e, which contain arithmetic and comparison operations, pairing and projections, and location dereference. Heaps H are maps from abstract locations ℓ to values v.

We use a number of auxiliary functions to access parts of $\mathsf{L^U}$ that we now explain. Function `locs(·)` returns the set of locations that are free in the argument of the function; that argument can be an expression, a list of functions or an attacker. Function $\texttt{names}(\cdot)$ returns the names of the functions defined in the argument, which can be a list of functions or a list of interfaces. Function $\texttt{fv}(\overline{F})$ returns the free variables in the bodies of the list of functions supplied as argument.

As explained in Section 2.1, safety properties are specified by monitors. Note that in place of the set $\{l\cdots\}$ of safety-relevant locations, the description of a monitor here (as well as a component above) contains a *single* location ℓ_{root}. The interpretation is that any location *reachable* in the heap starting from ℓ_{root} is relevant for safety. This set of locations can change as the program executes, and hence this is more flexible than statically specifying all of $\{l\cdots\}$ upfront. This representation of the set by a single location is made explicit in the following monitor rule:

$$\text{(L}^{\text{U}}\text{-Monitor Step)}\quad \frac{\begin{array}{c} M = (\{\sigma\cdots\}, \rightsquigarrow, \sigma_0, \ell_{root}, \sigma_c) \qquad M' = (\{\sigma\cdots\}, \rightsquigarrow, \sigma_0, \ell_{root}, \sigma_f) \\ (\sigma_c, H', \sigma_f) \in \rightsquigarrow \qquad H' \subseteq H \qquad \texttt{dom}(H') = \texttt{reach}(\ell_{root}, H) \end{array}}{M; H \rightsquigarrow M'}$$

$$\texttt{reach}(\ell', H) = \{\ell \mid \exists e.\ H \triangleright e \hookrightarrow\!\!\!\rightarrow \ell \land \ell \in \texttt{dom}(H) \land \texttt{locs}(e) = \ell'\}$$

Other than this small point, monitors, safety, robust safety and *RSC* are defined as in Section 2. In particular, a monitor and a component agree if they mention the same ℓ_{root} and an attacker is valid for a component C if its code and heap do not mention the ℓ_{root} location of C. Note that checking this condition is sufficient because whether A is a valid attacker is a *static condition*, checked before programs run. At this stage, ℓ_{root} does not point to any other location, so the check is sufficient. ℓ_{root} may grow to point to other locations, but these will be dynamically-generated, and thus the attacker cannot possibly mention them statically in its code.

$$M \frown C \overset{\text{def}}{=} (M = (\{\sigma\cdots\}, \rightsquigarrow, \sigma_0, \ell_{root}, \sigma_c)) \text{ and } (C = (\ell_{root}; \overline{F}; \overline{I}))$$

$$C \vdash A : \text{atk} \overset{\text{def}}{=} C = (\ell_{root}; \overline{F}; \overline{I}), A = H; \overline{F'} \text{ and } \ell_{root} \notin (\texttt{locs}(A))$$

The semantics of L^{U} relies on some auxiliary functions that we present in Figure 2 before presenting the semantics itself.

$$\text{(L}^{\text{U}}\text{-Jump-Internal)}\quad \frac{((f' \in \overline{I} \land f \in \overline{I}) \lor (f' \notin \overline{I} \land f \notin \overline{I}))}{\overline{I} \vdash f, f' : \text{internal}} \qquad \text{(L}^{\text{U}}\text{-Jump-IN)}\quad \frac{f \in \overline{I} \land f' \notin \overline{I}}{\overline{I} \vdash f, f' : \text{in}} \qquad \text{(L}^{\text{U}}\text{-Jump-OUT)}\quad \frac{f \notin \overline{I} \land f' \in \overline{I}}{\overline{I} \vdash f, f' : \text{out}}$$

$$\text{(L}^{\text{U}}\text{-Plug)}\quad \frac{\begin{array}{cc} A \equiv H; \overline{F}\,[\cdot] & C \equiv \ell_{root}; \overline{F'}; \overline{I} \\ \vdash C, \overline{F} : \text{whole} & \text{main} \in \texttt{names}(\overline{F}) \end{array}}{A\,[C] = \ell_{root}; H; \ell_{root} \mapsto 0; \overline{F}; \overline{F'}; \overline{I}} \qquad \text{(L}^{\text{U}}\text{-Whole)}\quad \frac{\begin{array}{c} C \equiv \ell_{root}; \overline{F'}; \overline{I} \qquad \texttt{fv}(\overline{F}) \cup \texttt{fv}(\overline{F'}) = \varnothing \\ \texttt{names}(\overline{F}) \cap \texttt{names}(\overline{F'}) = \varnothing \\ \texttt{names}(\overline{I}) \subseteq \texttt{names}(\overline{F}) \cup \texttt{names}(\overline{F'}) \end{array}}{\vdash C, \overline{F} : \text{whole}}$$

$$\text{(L}^{\text{U}}\text{-Initial State)}\quad \frac{P \equiv \ell_{root}; H; \overline{F}; \overline{I} \qquad C \equiv \ell_{root}; \overline{F}; \overline{I} \qquad \text{main}(x) \mapsto s; \text{return}; \in \overline{F}}{{}^{\cdot}_{0}\,(P) = C; H, \ell_{root} \mapsto 0 \triangleright (s[0\ /\ x])_{\text{main}}}$$

Fig. 2. Auxiliary rules. The first batch determines the direction of calls and returns. The second batch defines plugging a component with an attacker, when a whole program is whole and how to calculate the starting state of a program, which starts computing from the main function.

A program state ˙ includes the function bodies C, the heap H, a statement s being executed and a stack of function calls $\overline{f}$ (often omitted in the semantics rules for simplicity and explained in

Example 3.1). The initial state of a whole program is generated according to Rule L^{U}-Initial State. A program consisting of a component C and attacker-provided functions $\overline{\mathsf{F}}$ is whole (as defined by Rule L^{U}-Whole) if no function definition has free variables, if no function names is duplicated and if all import functions of the components are resolved. Whole programs are typically the result of plugging a component and an attacker together as in Rule L^{U}-Plug. The stack of function calls is used to populate judgements of the form $\overline{\mathsf{I}} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{internal}/\mathsf{in}/\mathsf{out}$ (Figure 2, top). These judgements determine whether calls and returns are $\mathsf{internal}$ (within the attacker or within the component), directed from the attacker to the component (in) or directed from the component to the attacker (out). This information is used to determine whether the semantics should generate a label (as in Rules EL^{U}-call to EL^{U}-retback) or no label (as in Rules EL^{U}-call-internal and EL^{U}-ret-internal) since internal calls should not be observable.

L^{U} has a big-step semantics for expressions that relies on evaluation contexts, a small-step semantics for statements that has labels λ and a semantics that accumulates labels in traces by omitting silent actions ϵ and concatenating the rest. These semantics follow the judgements below. The rules defining these judgments are presented in Figure 3:

$$\text{Expressions } \mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\rightarrow \mathsf{v} \qquad \text{Statements } \cdot \xrightarrow{\lambda} \cdot' \qquad \text{Traces } \cdot \xRightarrow{\overline{\alpha}} \cdot'$$

Unlike existing work on compositional compiler correctness which only relies on having the component [43], our semantics relies on having both the component and the context (i.e., a whole program).

Example 3.1 (Call semantics). To provide further insights on the semantics of this (and the following) language, this example shows the reduction for component $\mathsf{C}_{\mathsf{base}}$ below plugged with an attacker $\mathsf{A}_{\mathsf{base}}$ defining only function main (still below).

$$\mathsf{C}_{\mathsf{base}} = \ell_{\mathsf{root}}; \mathsf{skipten(x)} \mapsto \mathsf{if\ x >= 10\ then\ skip\ else\ call\ skipten\ x + 1; return;\ main}$$
$$\mathsf{A}_{\mathsf{base}} = \varnothing; \mathsf{main(x)} \mapsto \mathsf{call\ skipten\ 9; return;}$$

In the following, we indicate with C the component resulting by adding function main to the list of functions of $\mathsf{C}_{\mathsf{base}}$ and leaving the rest unmodified (as according to Rule L^{U}-Plug).

$$\mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright (\mathsf{call\ skipten\ 9; return;})_{\mathsf{main}}$$

$$\xrightarrow{\mathsf{call\ skipten\ 9\ \varnothing?}} \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright (\mathsf{if\ 9 >= 10\ then\ skip\ else\ call\ skipten\ 9 + 1; return;return;})_{\mathsf{main}\cdot\mathsf{skipten}}$$

since $\varnothing \triangleright \mathsf{9 >= 10} \hookrightarrow\!\!\!\rightarrow \mathsf{false}$

$$\longrightarrow \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright (\mathsf{call\ skipten\ 9 + 1; return;return;})_{\mathsf{main}\cdot\mathsf{skipten}}$$

since $\varnothing \triangleright \mathsf{9 + 1} \hookrightarrow\!\!\!\rightarrow \mathsf{10}$

(this call is internal to the component, so there is no label)

$$\longrightarrow \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright \begin{pmatrix} \mathsf{if\ 10 >= 10\ then\ skip\ else\ call\ skipten\ 10 + 1;} \\ \mathsf{return;return;return;} \end{pmatrix}_{\mathsf{main}\cdot\mathsf{skipten}\cdot\mathsf{skipten}}$$

since $\varnothing \triangleright \mathsf{10 >= 10} \hookrightarrow\!\!\!\rightarrow \mathsf{true}$

$$\longrightarrow \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright (\mathsf{skip; return;return;return;})_{\mathsf{main}\cdot\mathsf{skipten}\cdot\mathsf{skipten}}$$

$$\longrightarrow \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright (\mathsf{return;return;return;})_{\mathsf{main}\cdot\mathsf{skipten}\cdot\mathsf{skipten}}$$

(this return is internal to the component, so there is no label)

$$\longrightarrow \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright (\mathsf{skip; return;return;})_{\mathsf{main}\cdot\mathsf{skipten}}$$

$$\longrightarrow \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright (\mathsf{return;return;})_{\mathsf{main}\cdot\mathsf{skipten}}$$

$$\frac{\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{e}'}{\mathsf{H} \triangleright \mathsf{E}[\mathsf{e}] \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{E}[\mathsf{e}']}\ (\mathrm{EL^U\text{-}ctx}) \qquad \frac{}{\mathsf{H} \triangleright \mathsf{v} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}}\ (\mathrm{EL^U\text{-}val}) \qquad \frac{\ell \mapsto \mathsf{v} \in \mathsf{H}}{\mathsf{H} \triangleright !\ell \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}}\ (\mathrm{EL^U\text{-}dereference}) \qquad \frac{n \oplus n' = n''}{\mathsf{H} \triangleright \mathsf{n} \oplus \mathsf{n}' \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{n}''}\ (\mathrm{EL^U\text{-}op})$$

$$\frac{n \otimes n' = b}{\mathsf{H} \triangleright \mathsf{n} \otimes \mathsf{n}' \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{b}}\ (\mathrm{EL^U\text{-}comp}) \qquad \frac{}{\mathsf{H} \triangleright \langle \mathsf{v}, \mathsf{v}' \rangle.1 \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}}\ (\mathrm{EL^U\text{-}p1}) \qquad \frac{}{\mathsf{H} \triangleright \langle \mathsf{v}, \mathsf{v}' \rangle.2 \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}'}\ (\mathrm{EL^U\text{-}p2})$$

$$\frac{}{\mathsf{C}, \mathsf{H} \triangleright \mathsf{skip}; \mathsf{s} \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}}\ (\mathrm{EL^U\text{-}sequence}) \qquad \frac{\mathsf{C}, \mathsf{H} \triangleright \mathsf{s} \xrightarrow{\lambda} \mathsf{C}, \mathsf{H}' \triangleright \mathsf{s}'}{\mathsf{C}, \mathsf{H} \triangleright \mathsf{s}; \mathsf{s}'' \xrightarrow{\lambda} \mathsf{C}, \mathsf{H}' \triangleright \mathsf{s}'; \mathsf{s}''}\ (\mathrm{EL^U\text{-}step})$$

$$\frac{\begin{array}{c}\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v} \\ \mathsf{v} \equiv \mathsf{true} \Rightarrow \mathsf{s}'' = \mathsf{s} \qquad \mathsf{v} \equiv \mathsf{false} \Rightarrow \mathsf{s}'' = \mathsf{s}'\end{array}}{\mathsf{C}, \mathsf{H} \triangleright \mathsf{if\ e\ then\ s\ else\ s}' \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}''}\ (\mathrm{EL^U\text{-}if}) \qquad \frac{\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}}{\mathsf{C}, \mathsf{H} \triangleright \mathsf{let\ x = e\ in\ s} \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}[\mathsf{v} \,/\, \mathsf{x}]}\ (\mathrm{EL^U\text{-}letin})$$

$$\frac{\begin{array}{c}\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v} \\ \mathsf{H} = \mathsf{H}_1; \ell \mapsto \mathsf{v}'; \mathsf{H}_2 \qquad \mathsf{H}' = \mathsf{H}_1; \ell \mapsto \mathsf{v}; \mathsf{H}_2\end{array}}{\mathsf{C}, \mathsf{H} \triangleright \ell := \mathsf{e} \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H}' \triangleright \mathsf{skip}}\ (\mathrm{EL^U\text{-}update}) \qquad \frac{\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v} \qquad \ell \notin \mathrm{dom}(\mathsf{H})}{\begin{array}{l}\mathsf{C}, \mathsf{H} \triangleright \mathsf{let\ x = new\ e\ in\ s} \longrightarrow \\ \qquad \mathsf{C}, \mathsf{H}; \ell \mapsto \mathsf{v} \triangleright \mathsf{s}[\ell \,/\, \mathsf{x}]\end{array}}\ (\mathrm{EL^U\text{-}alloc})$$

$$\frac{\begin{array}{c}\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \qquad \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return}; \in \mathsf{C.funs} \\ \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}', \mathsf{f} : \mathsf{in} \qquad \mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}\end{array}}{\begin{array}{l}\mathsf{C}, \mathsf{H} \triangleright (\mathsf{call\ f\ e})_{\overline{\mathsf{f}'}} \xrightarrow{\mathtt{call\ f\ v\ H?}} \\ \qquad \mathsf{C}, \mathsf{H} \triangleright (\mathsf{s}; \mathsf{return}; [\mathsf{v} \,/\, \mathsf{x}])_{\overline{\mathsf{f}'}; \mathsf{f}}\end{array}}\ (\mathrm{EL^U\text{-}call}) \qquad \frac{\begin{array}{c}\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \qquad \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return}; \in \overline{\mathsf{F}} \\ \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}', \mathsf{f} : \mathsf{out} \qquad \mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}\end{array}}{\begin{array}{l}\mathsf{C}, \mathsf{H} \triangleright (\mathsf{call\ f\ e})_{\overline{\mathsf{f}'}} \xrightarrow{\mathtt{call\ f\ v\ H!}} \\ \qquad \mathsf{C}, \mathsf{H} \triangleright (\mathsf{s}; \mathsf{return}; [\mathsf{v} \,/\, \mathsf{x}])_{\overline{\mathsf{f}'}; \mathsf{f}}\end{array}}\ (\mathrm{EL^U\text{-}callback})$$

$$\frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \qquad \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{out}}{\mathsf{C}, \mathsf{H} \triangleright (\mathsf{return};)_{\overline{\mathsf{f}'}; \mathsf{f}} \xrightarrow{\mathtt{ret\ H!}} \mathsf{C}, \mathsf{H} \triangleright (\mathsf{skip})_{\overline{\mathsf{f}'}}}\ (\mathrm{EL^U\text{-}return}) \qquad \frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \qquad \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{in}}{\mathsf{C}, \mathsf{H} \triangleright (\mathsf{return};)_{\overline{\mathsf{f}'}; \mathsf{f}} \xrightarrow{\mathtt{ret\ H?}} \mathsf{C}, \mathsf{H} \triangleright (\mathsf{skip})_{\overline{\mathsf{f}'}}}\ (\mathrm{EL^U\text{-}retback})$$

$$\frac{\begin{array}{c}\overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{internal} \qquad \overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \\ \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return}; \in \mathsf{C.funs} \qquad \mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}\end{array}}{\begin{array}{l}\mathsf{C}, \mathsf{H} \triangleright (\mathsf{call\ f\ e})_{\overline{\mathsf{f}'}} \xrightarrow{\epsilon} \\ \qquad \mathsf{C}, \mathsf{H} \triangleright (\mathsf{s}; \mathsf{return}; [\mathsf{v} \,/\, \mathsf{x}])_{\overline{\mathsf{f}'}; \mathsf{f}}\end{array}}\ (\mathrm{EL^U\text{-}call\text{-}internal}) \qquad \frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \qquad \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{internal}}{\mathsf{C}, \mathsf{H} \triangleright (\mathsf{return};)_{\overline{\mathsf{f}'}; \mathsf{f}} \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \triangleright (\mathsf{skip})_{\overline{\mathsf{f}'}}}\ (\mathrm{EL^U\text{-}ret\text{-}internal})$$

$$\frac{\cdot \xrightarrow{\alpha} \cdot'}{\cdot \xRightarrow{\alpha} \cdot'}\ (\mathrm{EL^U\text{-}single}) \qquad \frac{\cdot \xrightarrow{\epsilon} \cdot'}{\cdot \Longrightarrow \cdot'}\ (\mathrm{EL^U\text{-}silent}) \qquad \frac{\cdot \xRightarrow{\overline{\alpha}} \cdot'' \qquad \cdot'' \xRightarrow{\overline{\alpha'}} \cdot'}{\cdot \xRightarrow{\overline{\alpha} \cdot \overline{\alpha'}} \cdot'}\ (\mathrm{EL^U\text{-}transitive})$$

Fig. 3. Semantics of $\mathsf{L^U}$. $\oplus$ includes $+, -, \times$. $\otimes$ includes $==, <, >$ etc; $[\mathsf{v} \,/\, \mathsf{x}]$ substitutes value v for variable x.

$$\begin{aligned} &\xrightarrow{\mathtt{ret}\ \varnothing!} \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright (\mathsf{skip}; \mathsf{return};)_{\mathsf{main}} \\ &\qquad \longrightarrow \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright (\mathsf{return};)_{\mathsf{main}} \\ &\qquad \longrightarrow \mathsf{C}; \varnothing; \ell_{\mathsf{root}} \triangleright \mathsf{skip}; \end{aligned}$$

3.2 The Target Language L^P

$$
\begin{array}{rlrl}
\textit{Components}\ C & ::= k_{root}; \overline{F}; \overline{I} & \textit{Heaps}\ H & ::= \varnothing \mid H; n \mapsto v : \eta \mid H; k \\
\textit{Expressions}\ e & ::= \cdots \mid !e \text{ with } e & \textit{Values}\ v & ::= n \in \mathbb{N} \mid \langle v, v \rangle \mid k
\end{array}
$$

$$
\textit{Statements}\ s ::= \cdots \mid \text{let } x = \text{hide } e \text{ in } s \mid \text{ifz } e \text{ then } s \text{ else } s \mid x := e \text{ with } e
$$

$$
\textit{Monitors}\ M ::= (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, k_{root}, \sigma_c) \quad \textit{Tags}\ \eta ::= \bot \mid k
$$

$$
\textit{Actions}\ \omega ::= \ \texttt{call}\ f\ v\ H? \mid \texttt{call}\ f\ v\ H! \mid \texttt{ret}\ H! \mid \texttt{ret}\ H?
$$

Fig. 4. Syntax of L^P. Elided bits $(\cdots)$ and omitted ones are the same as in L^U (Figure 1).

L^P is an untyped, imperative language that follows the structure of L^U and it has similar expressions and statements (Figure 4). However, there are critical differences (that make the compiler interesting). The main difference is that heap locations in L^P are concrete natural numbers. Upfront, an adversarial context can guess locations used as private state by a component and clobber them. To support hidden local state, a location can be "hidden" explicitly via the statement let x = hide e in s, which allocates a new *capability* k, an abstract token that grants access to the location n to which e points [72]. Subsequently, all reads and writes to n must be authenticated with the capability, so reading and writing a location take another parameter, the capability, as in $!e$ with e and $x := e$ with e. In both cases, the e after the with is the capability. Unlike locations, capabilities cannot be guessed. To make a location private, the compiler can make the capability of the location private. To bootstrap this hiding process, we assume that a component has one location that can only be accessed by it, a priori in the semantics (in our formalisation, we always focus on only one component and we assume that, for this component, this special location is at address 0).

L^P stores capabilities on the heap alongside locations, so a heap H contain both capabilities k as well as maps from natural numbers (locations) n to values v and a tag η. The tag η can be $\bot$, which means that n is globally available (not protected) or a capability k, which protects n. A globally available location can be freely read and written but one that is protected by a capability requires the same capability to be supplied at the time of read/write (Rule EL^P-assign, Rule EL^P-deref).

L^P has a big-step semantics for expressions, a labelled small-step semantics and a semantics that accumulates traces. These judgments follow similar judgements in the semantics of L^U (Figure 5).

A second difference between L^P and L^U is that L^P has no booleans, while L^U has them. This makes the compiler and the related proofs interesting, as discussed in the proof of Theorem 3.3.

In L^P, the locations of interest to a monitor are all those that can be reached from the address 0. Location 0 itself is protected with a capability k_{root} that is assumed to occur only in the code of the component in focus, so a component is defined as $C ::= k_{root}; \overline{F}; \overline{I}$. We can now give a precise definition of component-monitor agreement for L^P as well as a precise definition of attacker, which must care about the k_{root} capability. In the following, we use auxiliary function $\texttt{caps}(\cdot)$ to return the capabilities of the argument (analogously to how $\texttt{locs}(\cdot)$ returned the location of its argument).

$$
M \frown C \overset{\text{def}}{=} (M = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, k_{root}, \sigma_c)) \text{ and } (C = (k_{root}; \overline{F}; \overline{I}))
$$

$$
C \vdash A : \text{atk} \overset{\text{def}}{=} C.k_{root} \notin \texttt{caps}(A)
$$

$$\frac{\mathrm{n} \mapsto \mathrm{v} : \eta \in \mathrm{H} \qquad (\eta = \bot) \text{ or } (\eta = \mathrm{k} \text{ and } \mathrm{v}' = \mathrm{k})}{\mathrm{H} \triangleright !\mathrm{n} \text{ with } \mathrm{v}' \hookrightarrow\!\!\!\!\rightarrow \mathrm{H} \triangleright \mathrm{v}} \;(\mathrm{EL^P}\text{-deref})$$

$$\frac{\mathrm{H} \triangleright \mathrm{e} \hookrightarrow\!\!\!\!\rightarrow \mathrm{n} \qquad \mathrm{n} \equiv 0 \Rightarrow \mathrm{s}'' = \mathrm{s} \qquad \mathrm{n} \not\equiv 0 \Rightarrow \mathrm{s}'' = \mathrm{s}'}{\mathrm{C}, \mathrm{H} \triangleright \text{ifz e then s else s}' \xrightarrow{\epsilon} \mathrm{C}, \mathrm{H} \triangleright \mathrm{s}''} \;(\mathrm{EL^P}\text{-if})$$

$$\frac{\mathrm{H} = \mathrm{H}_1; \mathrm{n} \mapsto (\mathrm{v}', \eta) \qquad \mathrm{H} \triangleright \mathrm{e} \hookrightarrow\!\!\!\!\rightarrow \mathrm{v} \qquad \mathrm{H}' = \mathrm{H}; \mathrm{n}+1 \mapsto \mathrm{v} : \bot}{\mathrm{C}, \mathrm{H} \triangleright \text{let x = new e in s} \longrightarrow \mathrm{C}, \mathrm{H}' \triangleright \mathrm{s}[\mathrm{n}+1 \,/\, \mathrm{x}]} \;(\mathrm{EL^P}\text{-new})$$

$$\frac{\mathrm{H} \triangleright \mathrm{e} \hookrightarrow\!\!\!\!\rightarrow \mathrm{n} \qquad \mathrm{H} = \mathrm{H}_1; \mathrm{n} \mapsto \mathrm{v} : \bot; \mathrm{H}_2 \qquad \mathrm{k} \notin \mathrm{dom}(\mathrm{H}) \qquad \mathrm{H}' = \mathrm{H}_1; \mathrm{n} \mapsto \mathrm{v} : \mathrm{k}; \mathrm{H}_2; \mathrm{k}}{\mathrm{C}, \mathrm{H} \triangleright \text{let x = hide e in s} \longrightarrow \mathrm{C}, \mathrm{H}' \triangleright \mathrm{s}[\mathrm{k} \,/\, \mathrm{x}]} \;(\mathrm{EL^P}\text{-hide})$$

$$\frac{\begin{array}{c}\mathrm{H} \triangleright \mathrm{e} \hookrightarrow\!\!\!\!\rightarrow \mathrm{v} \qquad \mathrm{H} \triangleright \mathrm{e}' \hookrightarrow\!\!\!\!\rightarrow \mathrm{v}' \qquad \mathrm{H} = \mathrm{H}_1; \mathrm{n} \mapsto \_ : \eta; \mathrm{H}_2 \qquad \mathrm{H}' = \mathrm{H}_1; \mathrm{n} \mapsto \mathrm{v} : \eta; \mathrm{H}_2 \\ (\eta = \bot) \text{ or } (\eta = \mathrm{k} \text{ and } \mathrm{v}' = \mathrm{k})\end{array}}{\mathrm{C}, \mathrm{H} \triangleright \mathrm{n} := \text{e with e}' \longrightarrow \mathrm{C}, \mathrm{H}' \triangleright \text{skip}} \;(\mathrm{EL^P}\text{-assign})$$

Fig. 5. Expression and state semantics of $\mathrm{L^P}$. Omitted rules are the same as in $\mathrm{L^U}$ (Figure 3).

A monitor and compiler agree if they agree on $\mathrm{k_{root}}$. An attacker is valid if it does not contain $\mathrm{k_{root}}$ in its codebase a priori, though it may obtain $\mathrm{k_{root}}$ during its interaction with the component, if the component is not carefully written.

Remark. This language uses what is commonly referred to as 'data' capabilities, i.e., capabilities only for heap-allocated resources. Another kind of capabilities exist in the literature: 'code' capabilities, which grant the permission to jump to certain functions and execute their code. Since our programs do not have function pointers (nor higher-order functions), the code capabilities used and required by a program can be tracked statically, so we omit them entirely. If we extended our language with function pointers or higher-order functions, we would have to introduce code capabilities and pass said capabilities around in order to invoke the right function. We leave such an extension for future work.

3.3 Compiler from $\mathrm{L^U}$ to $\mathrm{L^P}$

We now present $[\![\cdot]\!]^{\mathrm{L^U}}_{\mathrm{L^P}}$, the compiler from $\mathrm{L^U}$ to $\mathrm{L^P}$, detailing how it uses the capabilities of $\mathrm{L^P}$ to achieve *RSC*. Then, we prove that $[\![\cdot]\!]^{\mathrm{L^U}}_{\mathrm{L^P}}$ attains *RSC*.

$[\![\cdot]\!]^{\mathrm{L^U}}_{\mathrm{L^P}}$ takes as input a $\mathrm{L^U}$ component C and returns a $\mathrm{L^P}$ component (Figure 6). The compiler performs a simple pass on the structure of functions, expressions and statements, using the information of the intended cross-language relation (β) to compile values. The only non-straightforward cases are the compilation of booleans and locations. Concerning the former, the compiler codes source booleans true to $\mathbf{0}$ and false to $\mathbf{1}$. Concerning the latter, each $\mathrm{L^U}$ location is encoded as a pair of a $\mathrm{L^P}$ location and the capability to access the location. Location update and dereference are compiled accordingly and thus project each pair to the location and the capability in order to use each part.

$$\left\llbracket \ell_{root}; \overline{F}; \overline{I} \right\rrbracket^{L^U}_{L^P} = \mathbf{k_{root}}; \left\llbracket \overline{F} \right\rrbracket^{L^U}_{L^P}; \left\llbracket \overline{I} \right\rrbracket^{L^U}_{L^P}$$

$$\llbracket f(x) \mapsto s; \text{return}; \rrbracket^{L^U}_{L^P} = \mathbf{f(x)} \mapsto \llbracket s \rrbracket^{L^U}_{L^P}; \mathbf{return;} \qquad \llbracket f \rrbracket^{L^U}_{L^P} = \mathbf{f}$$

$$\llbracket \text{true} \rrbracket^{L^U}_{L^P} = \mathbf{0} \qquad \llbracket \langle e_1, e_2 \rangle \rrbracket^{L^U}_{L^P} = \left\langle \llbracket e_1 \rrbracket^{L^U}_{L^P}, \llbracket e_2 \rrbracket^{L^U}_{L^P} \right\rangle$$

$$\llbracket \text{false} \rrbracket^{L^U}_{L^P} = \mathbf{1} \qquad \llbracket e.1 \rrbracket^{L^U}_{L^P} = \llbracket e \rrbracket^{L^U}_{L^P}\mathbf{.1}$$

$$\llbracket n \rrbracket^{L^U}_{L^P} = \mathbf{n} \qquad \llbracket e.2 \rrbracket^{L^U}_{L^P} = \llbracket e \rrbracket^{L^U}_{L^P}\mathbf{.2}$$

$$\llbracket x \rrbracket^{L^U}_{L^P} = \mathbf{x} \qquad \llbracket e \oplus e' \rrbracket^{L^U}_{L^P} = \llbracket e \rrbracket^{L^U}_{L^P} \oplus \llbracket e' \rrbracket^{L^U}_{L^P}$$

$$\llbracket \ell \rrbracket^{L^U}_{L^P} = \langle \mathbf{n}, \mathbf{v} \rangle \qquad \llbracket e \otimes e' \rrbracket^{L^U}_{L^P} = \llbracket e \rrbracket^{L^U}_{L^P} \otimes \llbracket e' \rrbracket^{L^U}_{L^P}$$

$$\llbracket !e \rrbracket^{L^U}_{L^P} = \mathbf{!}\llbracket e \rrbracket^{L^U}_{L^P}\mathbf{.1\ with\ }\llbracket e \rrbracket^{L^U}_{L^P}\mathbf{.2}$$

$$\llbracket \text{skip} \rrbracket^{L^U}_{L^P} = \mathbf{skip}$$

$$\llbracket s_u; s \rrbracket^{L^U}_{L^P} = \llbracket s_u \rrbracket^{L^U}_{L^P}; \llbracket s \rrbracket^{L^U}_{L^P}$$

$$\llbracket \text{call } f\ e \rrbracket^{L^U}_{L^P} = \mathbf{call\ f}\ \llbracket e \rrbracket^{L^U}_{L^P}$$

$$\llbracket \text{let } x = e \text{ in } s \rrbracket^{L^U}_{L^P} = \mathbf{let\ x =} \llbracket e \rrbracket^{L^U}_{L^P} \mathbf{\ in\ } \llbracket s \rrbracket^{L^U}_{L^P}$$

$$\left\llbracket \begin{array}{l} \text{if } e \text{ then } s_t \\ \quad \text{else } s_e \end{array} \right\rrbracket^{L^U}_{L^P} = \begin{array}{l} \mathbf{ifz}\ \llbracket e \rrbracket^{L^U}_{L^P}\ \mathbf{then}\ \llbracket s_t \rrbracket^{L^U}_{L^P} \\ \quad \mathbf{else}\ \llbracket s_e \rrbracket^{L^U}_{L^P} \end{array}$$

$$\left\llbracket \begin{array}{l} \text{let } x = \text{new } e \\ \quad \text{in } s \end{array} \right\rrbracket^{L^U}_{L^P} = \begin{array}{l} \mathbf{let\ x_{loc} = new}\ \llbracket e \rrbracket^{L^U}_{L^P}\ \mathbf{in} \\ \quad \mathbf{let\ x_{cap} = hide\ x_{loc}\ in} \\ \quad \mathbf{let\ x =} \left\langle \mathbf{x_{loc}, x_{cap}} \right\rangle \mathbf{\ in\ } \llbracket s \rrbracket^{L^U}_{L^P} \end{array}$$

$$\llbracket x := e \rrbracket^{L^U}_{L^P} = \begin{array}{l} \mathbf{let\ x1 = x.1\ in} \\ \quad \mathbf{let\ x2 = x.2\ in} \\ \quad \mathbf{x1 :=} \llbracket e \rrbracket^{L^U}_{L^P}\ \mathbf{with\ x2} \end{array}$$

Fig. 6. $\llbracket \cdot \rrbracket^{L^U}_{L^P}$, compilation of components and functions, expressions and statements from L^U to L^P.

This compiler solely relies on the capability abstraction of the target language as a defence mechanism to attain *RSC*. Unlike existing security-preserving compilers, $\llbracket \cdot \rrbracket^{L^U}_{L^P}$ needs neither dynamic checks nor other constructs that introduce runtime overhead to attain *RSC* [10, 35, 42, 59, 63].

3.3.1 *Proof of RSC.* $\llbracket \cdot \rrbracket^{L^U}_{L^P}$ attains *RSC* (Theorem 3.3). In order to set up this theorem, we need to instantiate the cross-language relation for values, which we write as $\approx_\beta$ here. The relation is parametrised by a partial bijection β : location $\times$ **natural number** $\times$ **tag** from source heap locations to target heap locations such that:

- if $(\ell_1, \mathbf{n}, \eta) \in \beta$ and $(\ell_2, \mathbf{n}, \eta) \in \beta$ then $\ell_1 = \ell_2$;
- if $(\ell, \mathbf{n_1}, \eta_1) \in \beta$ and $(\ell, \mathbf{n_2}, \eta_2) \in \beta$ then $\mathbf{n_1} = \mathbf{n_2}$ and $\eta_1 = \eta_2$.

The bijection determines when a source location and a target location (and its capability) are related.

On values, $\approx_\beta$ is defined as follows:

- $\text{true} \approx_\beta \mathbf{0}$;
- $\text{false} \approx_\beta \mathbf{n}$ for any $\mathbf{n} \neq \mathbf{0}$;
- $n \approx_\beta \mathbf{n}$;
- $\ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle$ if $(\ell, \mathbf{n}, \mathbf{k}) \in \beta$;
- $\ell \approx_\beta \langle \mathbf{n}, \_ \rangle$ if $(\ell, \mathbf{n}, \bot) \in \beta$;
- $\langle v_1, v_2 \rangle \approx_\beta \langle \mathbf{v_1}, \mathbf{v_2} \rangle$ if $v_1 \approx_\beta \mathbf{v_1}$ and $v_2 \approx_\beta \mathbf{v_2}$.

This relation is then used to define the heap, monitor state and action relations (Figure 7). Heaps are related, written $H \approx_\beta \mathbf{H}$, when locations related in β point to related values. States are related, written $\cdot \approx_\beta \Omega$, when they have related heaps. The action relation $\alpha \approx_\beta \omega$ is defined following the intuition of Section 2.2.

$$\text{(Heap relation)}\ \frac{H \approx_\beta H_1; H_2 \qquad \ell \approx_\beta \langle n, \eta \rangle \qquad v \approx_\beta v \qquad H = H_1; n \mapsto v : \eta; H_2}{H; \ell \mapsto v \approx_\beta H} \qquad \text{(Empty relation)}\ \frac{}{\varnothing \approx_\beta \overline{k}}$$

$$\text{(Related states – Whole)}\ \frac{\cdot = M; \overline{F}, \overline{F'}; \overline{I}; H \triangleright s \qquad \Omega = M; \overline{F}, \left[\!\left[\overline{F'}\right]\!\right]^{L^U}_{L^P}; \overline{I}; H \triangleright s \qquad M \approx_\beta M \qquad H \approx_\beta H}{\cdot \approx_\beta \Omega}$$

$$\text{(Call relation)}\ \frac{f \approx f \qquad v \approx_\beta v \qquad H \approx_\beta H}{\texttt{call } f\ v\ H? \approx_\beta \texttt{call } f\ v\ H?} \qquad \text{(Callback relation)}\ \frac{f \approx f \qquad v \approx_\beta v \qquad H \approx_\beta H}{\texttt{call } f\ v\ H! \approx_\beta \texttt{call } f\ v\ H!} \qquad \text{(Return relation)}\ \frac{H \approx_\beta H}{\texttt{ret } H! \approx_\beta \texttt{ret } H!}$$

$$\text{(Returnback relation)}\ \frac{H \approx_\beta H}{\texttt{ret } H? \approx_\beta \texttt{ret } H?} \qquad \text{(Epsilon relation)}\ \frac{}{\epsilon \approx_\beta \epsilon}$$

Fig. 7. Heap, state and action relations.

With this relation we state a backwards simulation lemma (Lemma 3.2) that is necessary for the *RSC* proof (and that can also yield whole program compiler correctness). Technically, since the semantics is deterministic, this lemma is derived from *forward* simulation, which is the same statement but with the source and target reductions swapped [47].

Lemma 3.2 (Backward simulation).

$$\textit{if } C, H \triangleright [\![s]\!]^{L^U}_{L^P} \xrightarrow{\lambda} C, H \triangleright [\![s']\!]^{L^U}_{L^P} \textit{ and } C, H \triangleright s \approx_\beta C, H \triangleright [\![s]\!]^{L^U}_{L^P} \textit{ and } \lambda \approx_\beta \lambda$$

$$\textit{then } C, H \triangleright s \xrightarrow{\lambda} C, H \triangleright s' \textit{ and } \exists \beta' \supseteq \beta.\, C, H \triangleright s' \approx_{\beta'} C, H \triangleright [\![s]\!]^{L^U}_{L^P}$$

The partial bijection β grows as we consider successive steps of program execution in our proof. For example, if executing let x = new e in s creates some source location ℓ, then executing its compiled counterpart will create some target location n and then protect that location with a fresh capability k. At this point we add (ℓ, n, k) to β.

Monitor Relation. In Section 2.2, we left the monitor relation abstract. Here, we define it for our two languages. Two monitors are related when they can *simulate* each other on related heaps. Given a monitor-specific relation $\sigma \approx \sigma$ on monitor states, we say that a relation $\mathcal{R}$ on source and target monitors is a *bisimulation* if the following hold whenever $M = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{root}, \sigma_c)$ and $M = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, k_{root}, \sigma_c)$ are related by $\mathcal{R}$:

(1) $\sigma_0 \approx \sigma_0$, and $\sigma_c \approx \sigma_c$, and
(2) For all β containing $(\ell_{root}, 0, k_{root})$ and all H, H with $H \approx_\beta H$:
  (a) $(\sigma_c, H, \_) \in \rightsquigarrow$ iff $(\sigma_c, H, \_) \in \rightsquigarrow$, and
  (b) $(\sigma_c, H, \sigma') \in \rightsquigarrow$ and $(\sigma_c, H, \sigma') \in \rightsquigarrow$ imply
  $(\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{root}, \sigma') \mathcal{R} (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, k_{root}, \sigma')$.

In words, $\mathcal{R}$ is a bisimulation only if $M \mathcal{R} M$ implies that M and M simulate each other on heaps related by *any* β that relates ℓ_{root} to 0. In particular, this means that neither M nor M can be sensitive to the *specific* addresses allocated during the run of the program. However, they can be sensitive to the "shape" of the heap or the values stored in the heap. Note that the union of any two bisimulations is a bisimulation. Hence, there is a largest bisimulation, which we denote as $\approx$. Intuitively, $\mathsf{M} \approx \mathbf{M}$ implies that M and $\mathbf{M}$ encode the same safety property (up to the relation $\approx_\beta$). With all the boilerplate for *RSC* in place, we state our main theorem.

Theorem 3.3 ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ attains *RSC*). $\vdash [\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} : RSC$

We outline our proof of Theorem 3.3, which relies on a backtranslation we denote $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$. Intuitively, $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$ takes a target trace $\overline{\omega}$ and builds a *set* of source contexts such that *one* of them when linked with the source program C, produces a related trace $\overline{\alpha}$ in the source (Theorem 3.6). In prior work, backtranslations return a single context [11, 12, 23, 30, 54, 59, 63]. This is because they all, explicitly or implicitly, assume that $\approx$ is injective from source to target. Under this assumption, the backtranslation is unique: a target value $\mathbf{v}$ will be related to at most one source value v. We do away with this assumption (e.g., the target value $\mathbf{0}$ is related to both source values $\mathsf{0}$ and true) and thus there can be multiple source values related to any given target value. This results in a set of backtranslated contexts, of which at least one will reproduce the trace as we need it as presented in Example 3.4.

Example 3.4 (Backtranslating a single context into a set). Consider a source component defining a single function $\mathsf{succ(x) \mapsto let\ y = x + 1\ in\ skip}$ and a target context linking against the compilation of that component. Assume the context defines $\mathbf{main(y) \mapsto call\ succ\ 0}$, which means that the trace semantics of the compiled component contains traces of the form $\mathbf{call\ succ\ 0\ \_?}$. Simply by reasoning at the target level, we cannot know whether $\mathbf{0}$ will be used as a boolean (e.g., in an $\mathbf{ifz\ e\ then\ e'\ else\ e''}$) or as a natural number (e.g., in a $\mathbf{x + 1}$). Thus, the backtranslation generates two contexts that call succ with both values that relate to $\mathbf{0}$:

$$\{\mathsf{main(y) \mapsto call\ succ\ true}; \mathsf{main(y) \mapsto call\ succ\ 0};\}$$

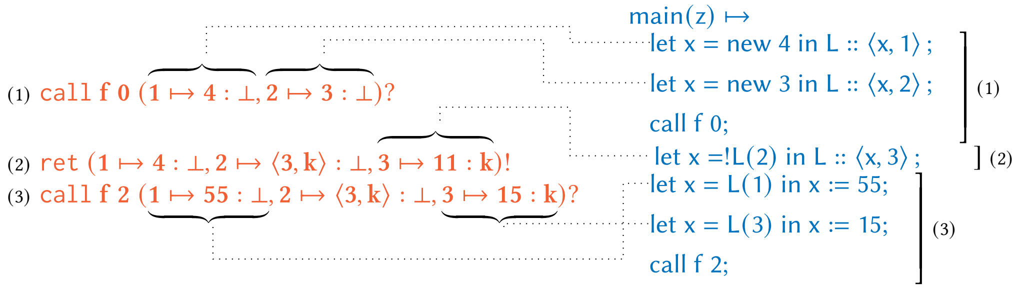
When the first main is linked with succ and they execute, the execution gets stuck inside succ: the $\mathsf{x + 1}$ expression is effectively $\mathsf{true + 1}$, which does not reduce. Since we know that the compiled program emits a !-decorated action, the same must be true in the source too. Thus, it is not possible that the execution gets stuck while executing code of the component and there must be another context that does not make the execution get stuck. In this case, that context is the one with the *second* implementation of main.

⊡

We bypass the lengthy technical setup for this proof and provide an informal description of why the backtranslation achieves what it is supposed to using an example (Example 3.5). We refer the interested reader to the accompanying technical report for full details on the backtranslation [65].

Notation. Example 3.5 needs to reason about lists of finite length whose elements are pairs, which are not a base type in our language. However, they can be easily encoded as sequences of pairs, with unit being the empty list. Thus, list $\mathsf{\langle 1, 1\rangle :: \langle 2, 2\rangle :: \langle 3, 3\rangle}$ is $\mathsf{\langle\langle 1, 1\rangle, \langle\langle 2, 2\rangle, \langle\langle 3, 3\rangle, unit\rangle\rangle\rangle}$. To maintain a lightweight notation, we use some syntactic sugar to encode lists of finite length to our language. We use metavariable L to indicate a pointer to a heap-allocated list of that form. Adding element v to list L is denoted as $\mathsf{L :: v}$; given that the content of L was some list list, adding an element amounts to making L point to $\mathsf{\langle v, list\rangle}$. Given an element $\mathsf{\langle v, n\rangle}$, we use notation $\mathsf{L(n)}$ to look that element up and return v (or unit if no element $\mathsf{\langle v, n\rangle}$ is in L). We can easily encode this lookup as a series of projections on the the list and then on each element of the list (note that our language is untyped, so this is possible).

Example 3.5 (Trace backtranslation). $\langle\langle\cdot\rangle\rangle^{\mathrm{L^P}}_{\mathrm{L^U}}$ first generates empty method bodies for all context methods called by the compiled component. Then it backtranslates each *action* on the given trace, generating code blocks that mimic that action and places that code inside the appropriate method body. The figure below shows an example trace on the left and the code blocks generated for each action in the trace on the right.



(1) $\texttt{call f 0}\ (\overbrace{1 \mapsto 4 : \bot}, \overbrace{2 \mapsto 3 : \bot})?$

(2) $\texttt{ret}\ (1 \mapsto 4 : \bot, 2 \mapsto \langle 3, k\rangle : \bot, \overbrace{3 \mapsto 11 : k})!$

(3) $\texttt{call f 2}\ (\underbrace{1 \mapsto 55 : \bot}, 2 \mapsto \langle 3, k\rangle : \bot, \underbrace{3 \mapsto 15 : k})?$

```
main(z) ↦
  let x = new 4 in L :: ⟨x, 1⟩ ;
  let x = new 3 in L :: ⟨x, 2⟩ ;          (1)
  call f 0;
  let x =!L(2) in L :: ⟨x, 3⟩ ;           (2)
  let x = L(1) in x := 55;
  let x = L(3) in x := 15;                (3)
  call f 2;
```

Backtranslated code maintains a support data structure at runtime, a list of locations denoted L that are known to the target. Locations are looked up in this list based on their second field n, which is their target-level address. Since we have access to the whole trace, we know how many locations we will add to L so we know its length. In order to backtranslate the first call, we need to set up the heap with the right values and then perform the call. In the diagram, dotted lines describe which source statement generates which part of the heap. The return only generates code that will update the list L to ensure that the context has access to all the locations it knows in the target too. In order to backtranslate the last call we look up the locations to be updated in L so we can ensure that when the call f 2 statement is executed, the heap is in the right state. ⊡

For the backtranslation to be used in the proof we need to prove its correctness, i.e., that $\langle\langle\overline{\omega}\rangle\rangle^{\mathrm{L^P}}_{\mathrm{L^U}}$ generates a context A that, together with C, generates a trace $\overline{\alpha}$ related to the given target trace $\overline{\omega}$. As before, the relatedness of actions (and of states) is stated with respect to a partial bijection β between source and target locations (and capabilities) that grows as the execution progresses.

Theorem 3.6 ($\langle\langle\cdot\rangle\rangle^{\mathrm{L^P}}_{\mathrm{L^U}}$ is correct).

$$\textit{if}\ \mathrm{A}\left[[\![\mathrm{C}]\!]^{\mathrm{L^U}}_{\mathrm{L^P}}\right] \overset{\overline{\omega}}{\Longrightarrow} \Omega\ \textit{then}\ \exists \mathrm{A} \in \langle\langle\overline{\omega}\rangle\rangle^{\mathrm{L^P}}_{\mathrm{L^U}}. \mathrm{A}\left[\mathrm{C}\right] \approx_{\beta} \mathrm{A}\left[[\![\mathrm{C}]\!]^{\mathrm{L^U}}_{\mathrm{L^P}}\right]\ \textit{and}\ \mathrm{A}\left[\mathrm{C}\right] \overset{\overline{\alpha}}{\Longrightarrow} \cdot$$
$$\textit{and}\ \exists \beta' \supseteq \beta. \overline{\alpha} \approx_{\beta'} \overline{\omega}\ \textit{and}\ \cdot \approx_{\beta'} \Omega.$$

This theorem immediately implies that $\vdash [\![\cdot]\!]^{\mathrm{L^U}}_{\mathrm{L^P}}$: *PF-RSC*, which, by Theorem 3.7 below, implies that $\vdash [\![\cdot]\!]^{\mathrm{L^U}}_{\mathrm{L^P}}$: *RSC*.

Theorem 3.7 (*PF-RSC* and *RSC* are equivalent for $[\![\cdot]\!]^{\mathrm{L^U}}_{\mathrm{L^P}}$).

$$\vdash [\![\cdot]\!]^{\mathrm{L^U}}_{\mathrm{L^P}} : \textit{PF-RSC} \iff \vdash [\![\cdot]\!]^{\mathrm{L^U}}_{\mathrm{L^P}} : \textit{RSC}$$

The intuition behind the proof of Theorem 3.7 follows the intuition we gave after Proposition 2.4. The only missing element in the proof is to demonstrate that related monitor states either both step or both get stuck on related source and target actions. We prove this by showing an invariant, namely that the monitor states always remain related. This follows from the rules of Figure 7. Finally, recall that the function `relevant`(α) returns just the heap of the action. This, with the relatedness of heaps, ensures that either both monitors step or both get stuck on related actions.

Example 3.8 (Compiling a secure program). To illustrate *RSC* at work, consider the following source component $\mathsf{C_a}$, which manages an account whose balance is security-relevant. The balance is stored in a location (ℓ_{root} that is tracked by the monitor). $\mathsf{C_a}$ provides functions to deposit to the account as well as to print the account balance.

$$\begin{aligned}
\mathsf{deposit(x)} &\mapsto \mathsf{let\ q{=}abs(x)\ in\ let\ amt = !\ell_{root}\ in\ \ell_{root} := amt + q}\\
\mathsf{balance(x)} &\mapsto \mathsf{let\ tmp{=}!\ell_{root}\ in\ skip}
\end{aligned}$$

$\mathsf{C_a}$ never leaks the sensitive location ℓ_{root} to an attacker. Additionally, an attacker has no way to decrement the amount of the balance since deposit only adds the absolute value $\mathsf{abs(x)}$ of its input x to the existing balance.

By compiling $\mathsf{C_a}$ with $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$, we obtain the following target program. For simplicity of reading, we provide a simplification of the compiled programs below the output of the compiler.

$$\begin{aligned}
\mathbf{deposit(x)} \mapsto\ & \mathbf{let\ q{=}abs(x)\ in}\\
& \mathbf{let\ amt{=}!\ \langle 0, k_{root}\rangle .1\ with\ \langle 0, k_{root}\rangle .2\ in}\\
& \mathbf{let\ l1{=}\langle 0, k_{root}\rangle .1\ in}\\
& \mathbf{let\ l2{=}\langle 0, k_{root}\rangle .2\ in}\\
& \mathbf{l1 := amt + q\ with\ l2}\\
\mathbf{balance(x)} \mapsto\ & \mathbf{let\ tmp{=}!\ \langle 0, k_{root}\rangle .1\ with\ \langle 0, k_{root}\rangle .2\ in\ skip}\\
& \text{simplified as}\\
\mathbf{deposit(x)} \mapsto\ & \mathbf{let\ q{=}abs(x)\ in}\\
& \mathbf{let\ amt{=}!0\ with\ k_{root}\ in}\\
& \mathbf{0 := amt + q\ with\ k_{root}}\\
\mathbf{balance(x)} \mapsto\ & \mathbf{let\ tmp{=}!0\ with\ k_{root}\ in\ skip}
\end{aligned}$$

Recall that location ℓ_{root} is mapped to location $\mathbf{0}$ and protected by the $\mathbf{k_{root}}$ capability. In the compiled code, while location $\mathbf{0}$ is freely computable by a target attacker, capability $\mathbf{k_{root}}$ is not. Since that capability is not leaked to an attacker by the code above, an attacker will not be able to tamper with the balance stored in location $\mathbf{0}$, even though it has direct access to $\mathbf{0}$. ⊡

Failing to Attain RSC. We now provide an insight of how could we detect whether a compiler is not *RSC*. In fact, if we had made a security-relevant mistake in the compiler, we would like one of the proofs to fall apart. Two ways to fail at attaining *RSC* come to mind: not using capabilities or leaking them; we explore the former for simplicity. Let us assume that the compiler does not protect compiled code with capabilities. Intuitively, compiled code is insecure because location $\mathbf{0}$ is freely accessible to the attacker, who can alter its content at will. In terms of proofs, this vulnerability manifests itself Theorem 3.6, whose proof would fail. Let us consider a valid target trace emitted by code compiled with the vulnerable compiler we discuss here. Such a trace would be one where the attacker changes the value of $\mathbf{0}$ as indicated below. There, the content of that location is reset to $\mathbf{3}$ between the return and the call (i.e., when the attacker executes).

$$\mathtt{call}\ f\ v\ 0 \mapsto 3 : \bot? \cdot \mathtt{ret}\ 0 \mapsto 1 : \bot! \cdot \mathtt{call}\ f\ v\ 0 \mapsto 3 : \bot?$$

No source attacker can do that, because location ℓ_{root} is not accessible to them. This would make the proof of Theorem 3.6 fail and reveal a hint that something that was supposed to be secure is now vulnerable.

4 *RSC* VIA BISIMULATION

If the source language has a verification system that enforces robust safety, proving that a compiler attains *RSC may* be simpler than that of Section 3 in some cases, as a backtranslation may not be needed at all. To demonstrate this, we consider a specific class of monitors, namely those that enforce type invariants on a specific set of locations. Our source language, L^τ, is similar to L^U but it has a type system that accepts only source programs whose traces the source monitor never rejects. Our target language is mostly unchanged. Our compiler $[\![\cdot]\!]^{L^\tau}_{L^\pi}$ is directed by typing derivations, and its proof of *RSC* relies on a cross-language invariant of program execution rather than a backtranslation. A second, independent goal of this section is to show that *RSC* is compatible with concurrency. Consequently, our source and target languages include constructs for forking threads.

4.1 The Source Language L^τ

$$\begin{array}{rlrl}
\textit{Components}\ \mathsf{C} ::= \text{´}; \overline{\mathsf{F}}; \overline{\mathsf{I}} & & \textit{Heaps}\ \mathsf{H} ::= \varnothing \mid \mathsf{H}; \ell \mapsto \mathsf{v} : \tau & \\
\textit{Types}\ \tau ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \tau \times \tau \mid \mathsf{Ref}\ \tau \mid \mathsf{UN} & & \textit{Envs.}\ \text{`} ::= \varnothing \mid \text{`}; (\mathsf{x} : \tau) & \\
\multicolumn{4}{c}{\textit{Superf. Types}\ \varphi ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{UN} \times \mathsf{UN} \mid \mathsf{Ref\ UN}} \\
\multicolumn{4}{c}{\textit{Statements}\ \mathsf{s} ::= \cdots \mid (\parallel \mathsf{s}) \mid \mathsf{endorse\ x = e\ as}\ \varphi\ \mathsf{in\ s}} \\
\textit{Monitors}\ \mathsf{M} ::= (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \text{´}, \sigma_c) & & \textit{Mon. Trans.} \rightsquigarrow ::= \varnothing \mid \rightsquigarrow; (\sigma, \sigma) & \\
\textit{Store Env.}\ \text{´} ::= \varnothing \mid \text{´}; (\ell : \tau) & & \textit{Processes}\ \pi ::= (\mathsf{s}) & \\
\textit{Soups}\ \text{˝} ::= \varnothing \mid \text{˝} \parallel \pi & & \textit{Prog. States}\ \text{˙} ::= \mathsf{C}, \mathsf{H} \triangleright \text{˝} &
\end{array}$$

Fig. 8. Syntax of L^τ. Elided and omitted elements are the same as in L^U (Figure 1).

L^τ extends L^U with concurrency, so it has a fork statement $(\parallel \mathsf{s})$, processes and process soups [21] and an extensive type system (Figure 8). Components define a set of safety-relevant locations ´, so and heaps carry type information. ´ also specifies a type for each safety-relevant location.

L^τ has an unconventional type system that enforces *robust type safety* [2, 15, 34, 37, 48, 62], which means that no context can cause the static types of sensitive heap locations to be violated at runtime. Using a special type UN that is described below, a program component statically partitions heap locations it deals with into those it cares about (sensitive or "trusted" locations) and those it does not care about ("untrusted" locations). Call a value *shareable* if only untrusted locations can be extracted from it using the language's elimination constructs. The type system then ensures that a program component only ever shares shareable values with the context. This ensures that the context cannot violate any invariants (including static types, which is what we care about in this section) of the trusted locations, since it can never gets direct access to them.

Type UN stands for "untrusted" or "shareable" and contains all values that can be passed to the context. Every type that is not a subtype of UN is implicitly trusted and cannot be passed to the context. Untrusted locations are explicitly marked UN at their allocation points in the program. Other types are deemed shareable via subtyping. Intuitively, a type is safe if values in it can only yield locations of type UN by the language's elimination constructs. For example, $\mathsf{UN} \times \mathsf{UN}$ is a subtype of UN. We write $\tau \vdash \circ$ to mean that τ is a subtype of UN.

Further, L^τ contains an *endorsement* statement ($\mathsf{endorse\ x = e\ as}\ \varphi\ \mathsf{in\ s}$) that dynamically checks the top-level constructor of a value of type UN and gives it a more precise superficial type φ [26]. This allows a program to safely inspect values coming from the context. It is similar to existing type casts [52] but it only inspects one structural layer of the value (this simplifies the compilation).

$\vdash C : \mathsf{UN}$	Component C is well-typed.	$C \vdash F : \tau$	Function F takes arguments of type τ.
$\acute{}, \grave{} \vdash \diamond$	Well-formed environments.	$\acute{}, \grave{} \vdash e : \tau$	Expression e has type τ in $\grave{}$ and $\acute{}$.
$\tau \vdash \circ$	Type τ is insecure.	$C, \acute{}, \grave{} \vdash s$	Statement s is well-typed in C, $\grave{}$ and $\acute{}$.

$$\text{(TL}^{\tau}\text{-bool-pub)}\ \frac{}{\mathsf{Bool} \vdash \circ} \qquad \text{(TL}^{\tau}\text{-nat-pub)}\ \frac{}{\mathsf{Nat} \vdash \circ} \qquad \text{(TL}^{\tau}\text{-pair-pub)}\ \frac{\tau \vdash \circ \quad \tau' \vdash \circ}{\tau \times \tau' \vdash \circ} \qquad \text{(TL}^{\tau}\text{-un-pub)}\ \frac{}{\mathsf{UN} \vdash \circ} \qquad \text{(TL}^{\tau}\text{-references-pub)}\ \frac{}{\mathsf{Ref\ UN} \vdash \circ}$$

$$\text{(TL}^{\tau}\text{-true)}\ \frac{\acute{}, \grave{} \vdash \diamond}{\acute{}, \grave{} \vdash \mathsf{true} : \mathsf{Bool}} \qquad \text{(TL}^{\tau}\text{-false)}\ \frac{\acute{}, \grave{} \vdash \diamond}{\acute{}, \grave{} \vdash \mathsf{false} : \mathsf{Bool}} \qquad \text{(TL}^{\tau}\text{-nat)}\ \frac{\acute{}, \grave{} \vdash \diamond}{\acute{}, \grave{} \vdash n : \mathsf{Nat}} \qquad \text{(TL}^{\tau}\text{-var)}\ \frac{x : \tau \in \grave{}}{\acute{}, \grave{} \vdash x : \tau}$$

$$\text{(TL}^{\tau}\text{-loc)}\ \frac{l : \tau \in \acute{}}{\acute{}, \grave{} \vdash l : \mathsf{Ref}\ \tau} \qquad \text{(TL}^{\tau}\text{-pair)}\ \frac{\begin{array}{c}\acute{}, \grave{} \vdash e_1 : \tau \\ \acute{}, \grave{} \vdash e_2 : \tau'\end{array}}{\acute{}, \grave{} \vdash \langle e_1, e_2 \rangle : \tau \times \tau'} \qquad \text{(TL}^{\tau}\text{-proj-1)}\ \frac{\acute{}, \grave{} \vdash e : \tau \times \tau'}{\acute{}, \grave{} \vdash e.1 : \tau} \qquad \text{(TL}^{\tau}\text{-proj-2)}\ \frac{\acute{}, \grave{} \vdash e : \tau \times \tau'}{\acute{}, \grave{} \vdash e.2 : \tau'}$$

$$\text{(TL}^{\tau}\text{-dereference)}\ \frac{\acute{}, \grave{} \vdash e : \mathsf{Ref}\ \tau}{\acute{}, \grave{} \vdash !e : \tau} \qquad \text{(TL}^{\tau}\text{-op)}\ \frac{\acute{}, \grave{} \vdash e : \mathsf{Nat} \quad \acute{}, \grave{} \vdash e' : \mathsf{Nat}}{\acute{}, \grave{} \vdash e \oplus e' : \mathsf{Nat}} \qquad \text{(TL}^{\tau}\text{-cmp)}\ \frac{\acute{}, \grave{} \vdash e : \mathsf{Nat} \quad \acute{}, \grave{} \vdash e' : \mathsf{Nat}}{\acute{}, \grave{} \vdash e \otimes e' : \mathsf{Bool}}$$

$$\text{(TL}^{\tau}\text{-coercion)}\ \frac{C, \acute{}, \grave{} \vdash e : \tau \quad \tau \vdash \circ}{C, \acute{}, \grave{} \vdash e : \mathsf{UN}}$$

$$\text{(TL}^{\tau}\text{-skip)}\ \frac{}{C, \acute{}, \grave{} \vdash \mathsf{skip}} \qquad \text{(TL}^{\tau}\text{-function-call)}\ \frac{\begin{array}{c}((f \in \mathsf{dom}(C.\mathsf{funs})) \lor (f \in \mathsf{dom}(C.\mathsf{intfs}))) \\ \acute{}, \grave{} \vdash e : \mathsf{UN}\end{array}}{\acute{}, \grave{} \vdash \mathsf{call}\ f\ e} \qquad \text{(TL}^{\tau}\text{-sequence)}\ \frac{\begin{array}{c}C, \acute{}, \grave{} \vdash s_u \\ C, \acute{}, \grave{} \vdash s\end{array}}{C, \acute{}, \grave{} \vdash s_u; s}$$

$$\text{(TL}^{\tau}\text{-letin)}\ \frac{\begin{array}{c}\acute{}, \grave{} \vdash e : \tau \\ C, \grave{}; x : \tau \vdash s\end{array}}{C, \acute{}, \grave{} \vdash \mathsf{let}\ x : \tau = e\ \mathsf{in}\ s} \qquad \text{(TL}^{\tau}\text{-assign)}\ \frac{\begin{array}{c}\acute{}, \grave{} \vdash x : \mathsf{Ref}\ \tau \\ \acute{}, \grave{} \vdash e' : \tau\end{array}}{C, \acute{}, \grave{} \vdash x := e'} \qquad \text{(TL}^{\tau}\text{-new)}\ \frac{\begin{array}{c}\acute{}, \grave{} \vdash e : \tau \\ C, \grave{}; x : \mathsf{Ref}\ \tau \vdash s\end{array}}{C, \acute{}, \grave{} \vdash \mathsf{let}\ x = \mathsf{new}_{\tau}\ e\ \mathsf{in}\ s}$$

$$\text{(TL}^{\tau}\text{-if)}\ \frac{\begin{array}{c}\acute{}, \grave{} \vdash e : \mathsf{Bool} \\ C, \acute{}, \grave{} \vdash s_t \quad C, \acute{}, \grave{} \vdash s_e\end{array}}{C, \acute{}, \grave{} \vdash \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_e} \qquad \text{(TL}^{\tau}\text{-fork)}\ \frac{C, \acute{}, \grave{} \vdash s}{C, \acute{}, \grave{} \vdash (\parallel s)} \qquad \text{(TL}^{\tau}\text{-endorse)}\ \frac{\begin{array}{c}\acute{}, \grave{} \vdash e : \mathsf{UN} \\ C, \acute{}, \grave{}; (x : \varphi) \vdash s\end{array}}{C, \acute{}, \grave{} \vdash \mathsf{endorse}\ x = e\ \mathsf{as}\ \varphi\ \mathsf{in}\ s}$$

Fig. 9. Typing judgements and rules of L^{τ}.

$$\text{(EL}^{\tau}\text{-endorse)}\ \frac{\begin{array}{c}H \triangleright e \hookrightarrow\!\!\!\!\rightarrow v \qquad \acute{}, \varnothing \vdash v : \varphi \\ \acute{} = \{\ell : \tau \mid \ell \mapsto v' : \tau \in H\}\end{array}}{C, H \triangleright \mathsf{endorse}\ x = e\ \mathsf{as}\ \varphi\ \mathsf{in}\ s \longrightarrow C, H \triangleright s[v / x]} \qquad \text{(EL}^{\tau}\text{-fork)}\ \frac{\begin{array}{c}\text{˝} = \text{˝}_1 \parallel (\parallel s); s' \parallel \text{˝}_2 \\ \text{˝}' = \text{˝}_1 \parallel \mathsf{skip}; s' \parallel \text{˝}_2 \parallel s\end{array}}{C, H \triangleright \text{˝} \longrightarrow C, H \triangleright \text{˝}'}$$

Fig. 10. Semantics of L^{τ}. Omitted elements are the same as in L^{U} (Figure 3).

The operational semantics of L^{τ} updates that of L^{U} to deal with concurrency and endorsement (Figure 10). For concurrency, the program state ˙ contains a soup (i.e., a multiset) ˝ of processes, where each process is a statement executing as in the program state for L^{U}, and a soup takes a step if any process in it does. The latter performs a runtime check on the endorsed value [67], which performs a syntactic check on a value given some superficial type φ. Superficial types φ only allow checking types “on the surface”, so pairs and references are not nested; in order to endorse a nested pair, multiple endorse statements must be used.

Monitors M check at runtime that the set of trusted heap locations ´ have values of their intended static types. Accordingly, the description of the monitor includes a list of trusted locations and their expected types (in the form of an environment ´). The type τ of any location in ´ must be trusted, so $\tau \nvdash \circ$. To facilitate the monitor’s checks, every heap location carries a type at runtime (in addition to a value). The monitor transitions should, therefore, be of the form $(\sigma, ´, \sigma')$, but since ´ never changes (it maps trusted locations to *static* types), we write the transitions as pairs of states only.

A monitor and a component agree if they have the same ´:

$$\mathsf{M} \frown \mathsf{C} \overset{\text{def}}{=} \mathsf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, ´, \sigma_c) \text{ and } \mathsf{C} = (´; \overline{\mathsf{F}}; \overline{\mathsf{I}})$$

Other definitions (safety, robust safety and actions) are as in Section 2.

Importantly, we show that a well-typed component generates traces that are always accepted by an agreeing monitor, so every component typed at UN is robustly safe.

Theorem 4.1 (Typability Implies Robust Safety in L^{τ}).

$$\textit{If } \vdash \mathsf{C} : \mathsf{UN} \textit{ and } \mathsf{C} \frown \mathsf{M} \textit{ then } \mathsf{M} \vdash \mathsf{C} : \mathsf{rs}$$

Richer Source Monitors. In L^{τ}, source language monitors only enforce the property of type safety on specific memory locations (robustly). This can be generalized substantially to enforce arbitrary invariants other than types on locations. The only requirement is to find a type system (e.g., based on refinements or Hoare logics) that can enforce robust safety in the source (for example, as in the work of Swasey *et al.* [76]). Our compilation and proof strategy should work with little modification. Another easy generalization is allowing the set of locations considered by the monitor to grow over time, as in Section 3.

4.2 The Target Language $\mathbf{L}^{\pi}$

Our target language, $\mathbf{L}^{\pi}$, extends the previous target language $\mathbf{L^{P}}$, with support for concurrency (forking, processes and process soups), atomic co-creation of a protected location and its protecting capability and for examining the top-level construct of a value according to a pattern $\mathbf{B}$ (Figure 11).

$$\textit{Statements } \mathbf{s} ::= \cdots \mid (\Vert\ \mathbf{s}) \mid \mathbf{let\ x = newhide\ e\ in\ s} \mid \mathbf{destruct\ x = e\ as\ B\ in\ s\ or\ s}$$

$$\textit{Patterns } \mathbf{B} ::= \mathbf{nat} \mid \mathbf{pair} \qquad \textit{Monitors } \mathbf{M} ::= (\{\boldsymbol{\sigma} \cdots\}, \rightsquigarrow, \boldsymbol{\sigma_0}, \mathbf{H_0}, \boldsymbol{\sigma_c})$$

$$\frac{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\rightarrow \mathbf{n}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{destruct\ x = e\ as\ nat\ in\ s\ or\ s'} \longrightarrow \mathbf{C}, \mathbf{H} \triangleright \mathbf{s}[\mathbf{n} \,/\, \mathbf{x}]}\ (\mathrm{E}\mathbf{L}^{\pi}\text{-destruct-nat})$$

$$\frac{\mathbf{H} = \mathbf{H_1}; \mathbf{n} \mapsto (\mathbf{v'}, \boldsymbol{\eta}) \quad \mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\rightarrow \mathbf{v} \quad \mathbf{k} \notin \mathsf{dom}(\mathbf{H}) \quad \mathbf{s'} = \mathbf{s}[\langle \mathbf{n+1}, \mathbf{k} \rangle \,/\, \mathbf{x}]}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ x = newhide\ e\ in\ s} \longrightarrow \mathbf{C}, \mathbf{H}; \mathbf{n+1} \mapsto \mathbf{v} : \mathbf{k}; \mathbf{k} \triangleright \mathbf{s'}}\ (\mathrm{E}\mathbf{L}^{\pi}\text{-new})$$

Fig. 11. Syntax and semantics of $\mathbf{L}^{\pi}$. Elided elements are either the same as $\mathbf{L^{P}}$ (Figures 4 and 5) or L^{τ} (Figure 8).

Monitors are also updated to consider a fixed set of locations (a heap part $\mathbf{H_0}$). Atomic co-creation of locations and capabilities is provided to match modern security architectures such as Cheri [81] (which implement capabilities at the hardware level). This atomicity is not strictly necessary and we prove that *RSC* is attained both by a compiler relying on it and by one that allocates a location and then protects it non-atomically. The former compiler (with this atomicity in the target) is a bit easier to describe, so we start with it (Section 4.3) before moving to the non-atomic one (Section 4.4).

4.3 Compiler from L^τ to L^π

The high-level structure of the compiler, $\llbracket\cdot\rrbracket^{L^\tau}_{L^\pi}$, is similar to that of our earlier compiler $\llbracket\cdot\rrbracket^{L^U}_{L^P}$ (Section 3.3). However, $\llbracket\cdot\rrbracket^{L^\tau}_{L^\pi}$ is defined by induction on the type derivation of the source component to be compiled. Most cases are a straightforward adaptation of the analogous cases from Figure 6, so we omit them. We show only a few instructional cases in Figures 12 and 13. When compiling a component we ensure that monitor-sensitive locations from ´ are allocated to related locations and initialised to valid values, i.e., values that respect the cross-language relation. Such a set up of heaps is denoted with ´ $\vdash_{\beta_0}$ H_0, whose details are in Rule Initial-heap (Figure 14). Intuitively, the domain of the safety-relevant heap must be related to the domain of ´. Additionally, the heap is populated with values v whose source-level counterparts have type τ; for locations, we do not allow cycles in memory for simplicity (Rule Initial-value). The most interesting cases of the compiler are are allocation and endorsement. The former explicitly uses type information to achieve security efficiently, protecting only those locations whose type is not UN. The latter performs a 1-level de-structuring of the value to be endorsed according to the expected superficial type φ.

$$\left\llbracket \frac{C \equiv ´;\overline{F};\overline{I} \qquad C \vdash \overline{F} : UN \qquad names(\overline{F}) \cap names(\overline{I}) = \varnothing \qquad ´ \vdash ok}{\vdash C : UN} \right\rrbracket^{L^\tau}_{L^\pi} = H_0; \left\llbracket\overline{F}\right\rrbracket^{L^\tau}_{L^\pi}; \left\llbracket\overline{I}\right\rrbracket^{L^\tau}_{L^\pi} \qquad \text{if } ´ \vdash_{\beta_0} H_0$$

$$\left\llbracket \frac{F \equiv f(x : UN) \mapsto s; return; \qquad C, ´; x : UN \vdash s \qquad \forall f \in fn(s), f \in dom(C.funs) \vee f \in dom(C.intfs)}{C \vdash F : UN} \right\rrbracket^{L^\tau}_{L^\pi} = f(x) \mapsto \llbracket C; ´; x : UN \vdash s\rrbracket^{L^\tau}_{L^\pi}; return;$$

$$\left\llbracket \frac{´, ` \vdash \diamond}{´, ` \vdash n : Nat} \right\rrbracket^{L^\tau}_{L^\pi} = n \qquad \left\llbracket \frac{x : \tau \in `}{´, ` \vdash x : \tau} \right\rrbracket^{L^\tau}_{L^\pi} = x \qquad \left\llbracket \frac{\ell : \tau \in ´}{´, ` \vdash \ell : \tau} \right\rrbracket^{L^\tau}_{L^\pi} = \langle n, v\rangle$$

$$\left\llbracket \frac{´, ` \vdash e : \tau \qquad \tau \vdash \circ}{´, ` \vdash e : UN} \right\rrbracket^{L^\tau}_{L^\pi} = \llbracket ´, ` \vdash e : \tau\rrbracket^{L^\tau}_{L^\pi}$$

$$\left\llbracket \frac{´, ` \vdash e : Nat \qquad ´, ` \vdash e' : Nat}{´, ` \vdash e \oplus e' : Nat} \right\rrbracket^{L^\tau}_{L^\pi} = \llbracket ´, ` \vdash e : Nat\rrbracket^{L^\tau}_{L^\pi} \oplus \llbracket ´, ` \vdash e' : Nat\rrbracket^{L^\tau}_{L^\pi}$$

$$\left\llbracket \frac{´, ` \vdash e : Nat \qquad ´, ` \vdash e' : Nat}{´, ` \vdash e \otimes e' : Bool} \right\rrbracket^{L^\tau}_{L^\pi} = \llbracket ´, ` \vdash e : Nat\rrbracket^{L^\tau}_{L^\pi} \otimes \llbracket ´, ` \vdash e' : Nat\rrbracket^{L^\tau}_{L^\pi}$$

$$\left\llbracket \frac{´, ` \vdash e : Ref\ \tau}{´, ` \vdash !e : \tau} \right\rrbracket^{L^\tau}_{L^\pi} = !\llbracket ´, ` \vdash e : Ref\ \tau\rrbracket^{L^\tau}_{L^\pi}.1 \text{ with } \llbracket ´, ` \vdash e : Ref\ \tau\rrbracket^{L^\tau}_{L^\pi}.2$$

Fig. 12. $\llbracket\cdot\rrbracket^{L^\tau}_{L^\pi}$, compilation of components, functions and expressions from L^τ to L^π (excerpts, omitted elements are analogous to their counterparts in Figure 6).

New Monitor Relation. As monitors have changed, we also need a new monitor relation $M \approx M$. Informally, a source and a target monitor are related if the target monitor can always step whenever the target heap satisfies the types specified in the source monitor's Δ (up to renaming by the partial bijection β_0).

$$\left\llbracket \frac{´,` \vdash e : \tau \qquad C, ´, `; x : \mathsf{Ref}\ \tau \vdash s}{C, ´, ` \vdash \mathsf{let}\ x = \mathsf{new}_\tau\ e\ \mathsf{in}\ s} \right\rrbracket^{L^\tau}_{L^\pi} = \begin{cases} \mathbf{let}\ xo\ = \mathbf{new}\ \llbracket ´,` \vdash e : \tau \rrbracket^{L^\tau}_{L^\pi} \\ \quad \mathbf{in\ let}\ x = \langle xo, 0 \rangle \\ \quad\ \mathbf{in}\ \llbracket C, ´, `; x : \mathsf{Ref}\ \tau \vdash s \rrbracket^{L^\tau}_{L^\pi} & \text{if } \tau = \mathsf{UN} \\ \mathbf{let}\ x\ = \mathbf{newhide}\ \llbracket ´,` \vdash e : \tau \rrbracket^{L^\tau}_{L^\pi} & \text{otherwise} \\ \quad \mathbf{in}\ \llbracket C, ´, `; x : \mathsf{Ref}\ \tau \vdash s \rrbracket^{L^\tau}_{L^\pi} \end{cases}$$

$$\left\llbracket \frac{\begin{array}{c} ´,` \vdash e : \mathsf{Bool} \\ C, ´, ` \vdash s_t \qquad C, ´, ` \vdash s_e \end{array}}{C, ´, ` \vdash \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_e} \right\rrbracket^{L^\tau}_{L^\pi} = \begin{array}{l} \mathbf{ifz}\ \llbracket ´,` \vdash e : \mathsf{Bool} \rrbracket^{L^\tau}_{L^\pi} \\ \ \mathbf{then}\ \llbracket C, ´, ` \vdash s_t \rrbracket^{L^\tau}_{L^\pi}\ \mathbf{else}\ \llbracket C, ´, ` \vdash s_e \rrbracket^{L^\tau}_{L^\pi} \end{array}$$

$$\left\llbracket \frac{C, ´, ` \vdash s_u \qquad C, ´, ` \vdash s}{C, ´, ` \vdash s_u; s} \right\rrbracket^{L^\tau}_{L^\pi} = \llbracket C, ´, ` \vdash s_u \rrbracket^{L^\tau}_{L^\pi}; \llbracket C, ´, `; `' \vdash s \rrbracket^{L^\tau}_{L^\pi}$$

$$\left\llbracket \frac{´,` \vdash e : \tau \qquad C, ´, `; x : \tau \vdash s}{C, ´, ` \vdash \mathsf{let}\ x : \tau = e\ \mathsf{in}\ s} \right\rrbracket^{L^\tau}_{L^\pi} = \mathbf{let}\ x{=}\llbracket ´,` \vdash e : \tau \rrbracket^{L^\tau}_{L^\pi}\ \mathbf{in}\ \llbracket C, ´, `; x : \tau \vdash s \rrbracket^{L^\tau}_{L^\pi}$$

$$\left\llbracket \frac{´,` \vdash x : \mathsf{Ref}\ \tau \qquad ´,` \vdash e : \tau}{C, ´, ` \vdash x := e} \right\rrbracket^{L^\tau}_{L^\pi} = \begin{array}{l} \mathbf{let}\ x_l = x.1\ \mathbf{in\ let}\ x_c = x.2 \\ \quad \mathbf{in}\ x_l := \llbracket ´,` \vdash e : \tau \rrbracket^{L^\tau}_{L^\pi}\ \mathbf{with}\ x_c \end{array}$$

$$\left\llbracket \frac{C, ´, ` \vdash s}{C, ´, ` \vdash (\parallel s)} \right\rrbracket^{L^\tau}_{L^\pi} = \left(\parallel \llbracket C, ´, ` \vdash s \rrbracket^{L^\tau}_{L^\pi} \right)$$

$$\left\llbracket \frac{´,` \vdash e : \mathsf{UN} \qquad C, ´, `; (x : \varphi) \vdash s}{C, ´, ` \vdash \mathsf{endorse}\ x = e\ \mathsf{as}\ \varphi\ \mathsf{in}\ s} \right\rrbracket^{L^\tau}_{L^\pi} =$$

$$\begin{cases} \mathbf{destruct}\ x\ = \llbracket ´,` \vdash e : \mathsf{UN} \rrbracket^{L^\tau}_{L^\pi}\ \mathbf{as\ nat\ in} \\ \quad \mathbf{ifz}\ x\ \mathbf{then}\ \llbracket C, ´, `; (x : \varphi) \vdash s \rrbracket^{L^\tau}_{L^\pi}\ \mathbf{else} \\ \qquad \mathbf{ifz}\ x - 1\ \mathbf{then}\ \llbracket C, ´, `; (x : \varphi) \vdash s \rrbracket^{L^\tau}_{L^\pi}\ \mathbf{else\ wrong} & \text{if } \varphi = \mathsf{Bool} \\ \ \mathbf{or\ wrong} \\ \mathbf{destruct}\ x\ = \llbracket ´,` \vdash e : \mathsf{UN} \rrbracket^{L^\tau}_{L^\pi}\ \mathbf{as\ nat\ in}\ \llbracket C, ´, `; (x : \varphi) \vdash s \rrbracket^{L^\tau}_{L^\pi} & \text{if } \varphi = \mathsf{Nat} \\ \ \mathbf{or\ wrong} \\ \mathbf{destruct}\ x\ = \llbracket ´,` \vdash e : \mathsf{UN} \rrbracket^{L^\tau}_{L^\pi}\ \mathbf{as\ pair\ in}\ \llbracket C, ´, `; (x : \varphi) \vdash s \rrbracket^{L^\tau}_{L^\pi} & \text{if } \varphi = \mathsf{UN} \times \mathsf{UN} \\ \ \mathbf{or\ wrong} \\ \mathbf{destruct}\ x\ = \llbracket ´,` \vdash e : \mathsf{UN} \rrbracket^{L^\tau}_{L^\pi}\ \mathbf{as\ pair\ in}\ !x.1\ \mathbf{with}\ x.2; \llbracket C, ´, `; (x : \varphi) \vdash s \rrbracket^{L^\tau}_{L^\pi} & \text{if } \varphi = \mathsf{Ref}\ \mathsf{UN} \\ \ \mathbf{or\ wrong} \end{cases}$$

Fig. 13. $\llbracket \cdot \rrbracket^{L^\tau}_{L^\pi}$, compilation of statements from L^τ to L^π (excerpts, omitted elements are analogous to their counterparts in Figure 6).

We write $\vdash H : ´$ to mean that for each location $\ell \in ´$, we have that $´; \varnothing \vdash H(\ell) : ´(\ell)$: i.e., the contents of H are well-typed according to $´$. Given a partial bijection β from source to target locations, we say that a target monitor $M = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, H_0, \sigma_c)$ is good, written $\vdash M : \beta, ´$, if for all $\sigma \in \{\sigma \cdots\}$ and all $H \approx_\beta H$ such that $\vdash H : ´$, there is a σ' such that $(\sigma, H, \sigma') \in \rightsquigarrow$. For a fixed partial bijection β_0 between the domains of $´$ and H_0, we say that the source monitor M and the target monitor M are related, written $M \approx M$, if $\vdash M : \beta_0, ´$ for the $´$ in M. With this setup, we define *RSC* as in Section 2. Our main theorem is that $\llbracket \cdot \rrbracket^{L^\tau}_{L^\pi}$ attains *RSC* under this definition.

Theorem 4.2 (Compiler $\llbracket \cdot \rrbracket^{L^\tau}_{L^\pi}$ attains *RSC*). $\vdash \llbracket \cdot \rrbracket^{L^\tau}_{L^\pi} : RSC$

$$\frac{´ \vdash \mathbf{H} \qquad ´, \mathbf{H} \vdash_{\beta} \mathbf{v} \colon \tau \qquad \ell \approx_{\beta} \langle \mathbf{n}, \mathbf{k} \rangle}{´, \ell : \tau \vdash_{\beta} \mathbf{H}; \mathbf{n} \mapsto \mathbf{v} : \mathbf{k}} \text{ (Initial-heap)}$$

$$\frac{\begin{array}{c} (\tau \equiv \mathsf{Bool} \land \mathbf{v} \equiv \mathbf{0}) \qquad \lor \qquad (\tau \equiv \mathsf{Nat} \land \mathbf{v} \equiv \mathbf{0}) \qquad \lor \\ (\tau \equiv \mathsf{Ref}\ \tau \land \mathbf{v} \equiv \mathbf{n'} \land \mathbf{n'} \mapsto \mathbf{v'} : \mathbf{k'} \in \mathbf{H} \land \ell' \approx_{\beta} \langle \mathbf{n'}, \mathbf{k'} \rangle \land \ell : \tau \in ´, ´, \mathbf{H} \vdash \mathbf{v'} \colon \tau) \qquad \lor \\ (\tau \equiv \tau_1 \times \tau_2 \land \mathbf{v} \equiv \langle \mathbf{v_1}, \mathbf{v_2} \rangle \land ´, \mathbf{H} \vdash \mathbf{v_1} \colon \tau_1 \land ´, \mathbf{H} \vdash \mathbf{v_2} \colon \tau_2) \end{array}}{´, \mathbf{H} \vdash_{\beta} \mathbf{v} \colon \tau} \text{ (Initial-value)}$$

Fig. 14. Initialisation of the safety-relevant target heap based on the source typing environment.

To prove that $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ attains *RSC* we do not rely on a backtranslation and we show *RSC* as of Definition 2.2 instead of the property-free version. Here, we know statically which locations can be monitor-sensitive: they must all be trusted, i.e., must have a type τ satisfying $\tau \nvdash \circ$. Using this, we set up a simple cross-language relation (later indicated as $\approx_{\beta}$) and show it to be an invariant on runs of source and compiled target states. Like previous relations, this relation is also indexed by a partial bijection β. The relation captures the following:

- Heaps (both source and target) can be partitioned into two parts, a *trusted* part and an *untrusted* part;
- The trusted source heap contains only locations whose type is trusted ($\tau \nvdash \circ$);
- The trusted target heap contains only locations related to trusted source locations and these point to related values; more importantly, every trusted target location is protected by a capability;
- In the target, any capability protecting a trusted location does not occur in attacker code, nor is it stored in an untrusted heap location.

We need to prove that this relation is preserved by reductions both in compiled and in attacker code. The former follows from the proof of source robust safety (Theorem 4.1).

The latter is formalised in Lemma 4.3 below and it is simple to prove. Since all trusted locations are protected with capabilities, attackers have no access to trusted locations, and capabilities are unforgeable and unguessable (by the semantics of $\mathbf{L}^{\pi}$). At this point, knowing that the monitors at hand are related, and that source traces are always accepted by the considered source monitors, we can conclude that target traces are always accepted by the considered target monitors too. Note that this kind of an argument requires all compilable source programs to be robustly safe and is, therefore, impossible for our first compiler $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$. Overall, avoiding the backtranslation results in a proof much simpler than that of Section 3.

In order to state the lemma discussed above, we rely on notation $\mathsf{C} \vdash_{\mathsf{att}} ″ \xrightarrow{\lambda} ″'$, which states that the reduction taking place occurs in attacker code. This can be easily defined by observing the stack of functions (explained in Example 3.1) and controlling whether the executing function is defined by the attacker or not. Note that unlike before, we do not need to grow the partial bijection here because it is used to track locations as used by the compiled code and not by the attacker.

Lemma 4.3 (Attacker actions preserve relatedness).

$$\textit{if } \mathsf{C}, \mathsf{H} \triangleright ″ \xrightarrow{\lambda} \mathsf{C}, \mathsf{H'} \triangleright ″' \textit{ and } \mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi} \xrightarrow{\lambda} \mathbf{C}, \mathbf{H'} \triangleright \mathbf{\Pi'} \textit{ and } \mathsf{C}, \mathsf{H} \triangleright ″ \approx_{\beta} \mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi}$$

$$\textit{and } \mathsf{C} \vdash_{\mathsf{att}} ″ \xrightarrow{\lambda} ″' \textit{ and } \mathbf{C} \vdash_{\mathsf{att}} \mathbf{\Pi} \xrightarrow{\lambda} \mathbf{\Pi'} \textit{ then } \mathsf{C}, \mathsf{H'} \triangleright ″' \approx_{\beta} \mathbf{C}, \mathbf{H'} \triangleright \mathbf{\Pi'}$$

4.4 Non-Atomic Allocation of Capabilities

The compiler of Figure 12 uses a new target language construct, **newhide**, that simultaneously allocates a new location and protects it with a capability. This atomic construct is what certain capability machines provide, and it simplifies our proof of security in the concurrent setting at hand. If allocation and protection were not atomic, then a concurrent adversary thread could protect a location that had just been allocated and acquire the capability to it before the allocating thread could do so. This would break the cross-language relation we use in our proof. However, note that this is not really an attack since it does not give the adversary any additional power to violate the safety property enforced by the monitor. The reason is that the thread allocating the location gets stuck when it tries to acquire the capability itself (since the adversary obtained the capability), and, by the design of our compiler, it will not try to use the location before obtaining the capability. Consequently, it is possible to do away with the **newhide** construct for compiling allocation. Figure 15 shows how compilation can also be done using the $\mathbf{let}\ \mathbf{x_k} = \mathbf{hide}\ \mathbf{x}\ \mathbf{in} \ldots$ construct of Section 3.

$$\left[\!\left[\frac{´, ` \vdash \mathsf{e} : \tau \qquad \mathsf{C}, ´, `; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}}{\mathsf{C}, ´, ` \vdash \mathsf{let}\ \mathsf{x} = \mathsf{new}_\tau\ \mathsf{e}\ \mathsf{in}\ \mathsf{s}} \right]\!\right]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \begin{cases} \mathbf{let}\ \mathbf{x_t} = \mathbf{new}\ \mathbf{0}\ \mathbf{in} \\ \quad \mathbf{let}\ \mathbf{x_k} = \mathbf{hide}\ \mathbf{x_t}\ \mathbf{in} \\ \quad \mathbf{let}\ \mathbf{x_c} = [\![´, ` \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\ \mathbf{in} \\ \quad \mathbf{x_t} := \mathbf{x_c}\ \mathbf{with}\ \mathbf{x_k}; \\ \quad \mathbf{let}\ \mathbf{x} = \langle \mathbf{x_t}, \mathbf{x_k} \rangle\ \mathbf{in} \\ \quad [\![\mathsf{C}, ´, `; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} \end{cases} \text{if } \tau \neq \mathsf{UN}$$

Fig. 15. Non-atomic implementation of capability allocation, only the interesting case for $\tau \neq \mathsf{UN}$ is reported, the other is analogous to the one in Figure 12.

The price to pay is a slightly more involved cross-language relation, which must relate states where either (i) the heaps are partitioned as before, or (ii) the target execution is stuck trying to acquire a capability for a location that should be trusted.

We refer the interested reader to the accompanying technical report for details of the new relation and the proof that this alternative compiler also attains *RSC* [65].

5 *RSC* RELYING ON TARGET MEMORY ISOLATION

Both compilers presented so far used a capability-based target language. To avoid giving the false impression that *RSC* is only useful for this kind of a target, we show here how to attain *RSC* when the protection mechanism in the target is completely different. We consider a new target language, L^I, which does not have capabilities, but instead offers *coarse-grained* memory isolation based on *enclaves* (Section 5.1). This mechanism is supported (in hardware) in mainstream x86-64 and ARM CPUs (Intel calls this SGX [49]; ARM calls it TrustZone [77]). It is also straightforward to implement purely in software using any VM-based, process-based, or in-process isolation technique. We present a compiler $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ from our last source language L^τ to L^I and prove that it attains *RSC* (Section 5.2).

5.1 L^I, a Target Language with Coarse-Grained Memory Isolation

Language L^I replaces $\mathbf{L}^\pi$'s capabilities with a simple security abstraction called an enclave. An enclave is a collection of code and memory locations, with the properties that: (a) only code within the enclave can access the memory locations of the enclave, and (b) code from outside can transfer control only to designated entry points in the enclave's code. For simplicity, L^I supports only one enclave. Generalizing this to many enclaves is straightforward, but not necessary for our purposes.

To model the enclave, L^I components carry additional information $\overline{E}$, the list of functions that reside in the enclave. Only functions that are listed in $\overline{E}$ can create, read and write locations in the enclave, using statements and expressions. Locations in L^I are *integers* (not natural numbers). By convention, non-negative locations are outside the enclave (accessible from any function), while negative locations are inside the enclave (accessible only from functions in $\overline{E}$).

$$\begin{array}{lll}
\textit{Components } C ::= H_0; \overline{F}; \overline{I}; \overline{E} & \textit{Enclave funcs. } E ::= f & \textit{Heaps } H ::= \varnothing \mid H; n \mapsto v \\
\textit{Values } v ::= n \in \mathbb{Z} \mid \langle v, v \rangle \mid k & \textit{Expressions } e ::= \cdots \mid !e & \\
\multicolumn{3}{l}{\textit{Statements } s ::= \cdots \mid x := e \mid \textit{let } x = \textit{new } e \textit{ in } s \mid \textit{let } x = \textit{newiso } e \textit{ in } s}
\end{array}$$

Fig. 16. Syntax of L^I. Elided and omitted elements are the same as in $\mathbf{L^P}$ (Figure 4) or $\mathbf{L}^\pi$ (Figure 11).

The semantics (Figure 17) are almost those of $\mathbf{L}^\pi$, but the expression semantics change to $C; H; f \triangleright e \hookrightarrow\!\!\!\!\rightarrow v$, recording which function f is currently executing. The operational rule for any memory operation checks that either the access is to a location outside the enclave or that $f \in \overline{E}$ (formalised by $C \vdash f : prog$). Allocating a protected location (Rule EL^I-isolate) is done with respect to location n, which is the smallest allocated location. This way, the new location $n - 1$ will be for sure in the domain of the enclave. Monitors of L^I are the same as those of $\mathbf{L}^\pi$.

$$\frac{n \mapsto v \in H \qquad (n \geq 0) \text{ or } (n < 0 \text{ and } C \vdash f : prog)}{C; H; f \triangleright !n \hookrightarrow\!\!\!\!\rightarrow v}\ (\text{E}L^I\text{-deref})$$

$$\frac{H = n \mapsto \_; H_1 \qquad C; H; f \triangleright e \hookrightarrow\!\!\!\!\rightarrow v \qquad C \vdash f : prog}{C, H \triangleright (\textit{let } x = \textit{newiso } e \textit{ in } s)_{\overline{f};f} \xrightarrow{\epsilon} C, n-1 \mapsto v; H \triangleright (s[n-1 \,/\, x])_{\overline{f};f}}\ (\text{E}L^I\text{-isolate})$$

$$\frac{\begin{array}{c} C; H; f \triangleright e \hookrightarrow\!\!\!\!\rightarrow v \qquad H = H_1; n \mapsto \_; H_2 \qquad H' = H_1; n \mapsto v; H_2 \\ (n \geq 0) \text{ or } (n < 0 \text{ and } C \vdash f : prog) \end{array}}{C, H \triangleright (n := e)_{\overline{f};f} \xrightarrow{\epsilon} C, H' \triangleright (\textit{skip})_{\overline{f};f}}\ (\text{E}L^I\text{-assign})$$

Fig. 17. Semantics of L^I. Omitted rules are as in $\mathbf{L^P}$ (Figure 5) and L^τ (Figure 10).

5.2 Compiler from L^τ to L^I

The high-level structure of the compiler $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ is similar to that of $[\![\cdot]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$ from Section 4.3 (Figure 18). Compiler $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ ensures that all the (and only the) functions of the (trusted) component we write are part of the enclave, i.e., constitute $\overline{E}$. Additionally, the compiler populates the safety-relevant heap H_0 based on the information in $'$ according to bijection φ. This is captured by the judgement $' \vdash_\varphi H_0$, whose details are in Figure 19 and they follow the same intuition of Figure 14. Compiler, $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ also ensures that trusted locations are stored in the enclave. As before, the compiler relies on typing information for this. Locations whose types are shareable (subtypes of UN) are placed

$$\left[\!\!\left[\frac{\begin{array}{c}\mathsf{C} \equiv ´;\overline{\mathsf{F}};\overline{\mathsf{I}} \quad \mathsf{C} \vdash \overline{\mathsf{F}} : \mathsf{UN} \quad ´ \vdash \mathsf{ok} \\ \mathsf{names}(\overline{\mathsf{F}}) \cap \mathsf{names}(\overline{\mathsf{I}}) = \varnothing\end{array}}{\vdash \mathsf{C} : \mathsf{UN}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}} = H_0; \left[\!\!\left[\overline{\mathsf{F}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}; \left[\!\!\left[\overline{\mathsf{I}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}; \mathsf{dom}(\overline{\mathsf{F}}) \qquad \text{if } ´ \vdash_{\varphi} H_0$$

$$\left[\!\!\left[\frac{\begin{array}{c}\mathsf{C}, ´, ` \vdash \mathsf{e} : \tau \\ \mathsf{C}, ´, `; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}\end{array}}{\begin{array}{l}\mathsf{C}, ´, ` \vdash \\ \quad \mathsf{let}\ \mathsf{x} = \mathsf{new}_{\tau}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s}\end{array}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}} = \begin{cases} \begin{array}{l} \mathit{let}\ x\ = \mathit{new}\ [\![´, ` \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{L^{I}} \\ \quad \mathit{in}\ [\![\mathsf{C}, ´, `; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{L^{I}} \end{array} & \text{if } \tau = \mathsf{UN} \\ \begin{array}{l} \mathit{let}\ x\ = \mathit{newiso}\ [\![´, ` \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{L^{I}} \\ \quad \mathit{in}\ [\![\mathsf{C}, ´, `; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{L^{I}} \end{array} & \text{else} \end{cases}$$

Fig. 18. $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$, compilation of components and statements from L^{τ} to L^{I} (excerpts, missing bits are adaptations of those bits from Figures 6 and 12).

outside the enclave while those that are trusted (not subtypes of UN) are placed inside (second rule in Figure 18).

$$\text{(Initial-heap)}\quad \frac{´ \vdash H \qquad ´, H \vdash v : \tau \qquad \ell \approx_{\varphi} n}{´, \ell : \tau \vdash_{\varphi} H; n \mapsto v}$$

$$\text{(Initial-value)}\quad \frac{\begin{array}{c}(\tau \equiv \mathsf{Bool} \land v \equiv 0) \qquad \lor \qquad (\tau \equiv \mathsf{Nat} \land v \equiv 0) \qquad \lor \\ (\tau \equiv \mathsf{Ref}\ \tau \land v \equiv n' \land n' \mapsto v' \in H \land \ell' \approx_{\varphi} n' \land \ell : \tau \in ´, ´, H \vdash v' : \tau) \qquad \lor \\ (\tau \equiv \tau_1 \times \tau_2 \land v \equiv \langle v_1, v_2 \rangle \land ´, H \vdash v_1 : \tau_1 \land ´, H \vdash v_2 : \tau_2)\end{array}}{´, H \vdash_{\varphi} v : \tau}$$

Fig. 19. Initialisation of the safety-relevant target heap based on the source typing environment.

For this compiler we need a different partial bijection that drops capabilities and considers integers instead of natural numbers. We indicate such a bijection with φ. Its type is $\ell \times n$, but it has the same properties as β in Section 3.3.1.

The cross-language relation $\approx$ is mostly unchanged. The only change is for relating locations, as defined below:

- $\ell \approx_{\varphi} n$ if $(\ell, n) \in \varphi$

Our main theorem is that $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$ attains *RSC*.

THEOREM 5.1 (COMPILER $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$ ATTAINS *RSC*). $\vdash [\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}} : \mathit{RSC}$

The intuition behind the proof is simple: all trusted locations (including safety-relevant locations) are in the enclave and adversarial code cannot tamper with them. The proof follows the idea of the proof of Theorem 4.2: we build a cross-language relation, which we show to be an invariant on executions of source and corresponding compiled programs. The only change is that every location in the *trusted target heap* is isolated in the enclave.

6 FULLY ABSTRACT COMPILATION

Our next goal is to compare *RSC* to fully abstract compilation (or *FAC*) at an intuitive level. We first define *FAC* (Section 6.1). Then, we present a series of examples of how *FAC* may result in inefficiencies in compiled code (Section 6.2). Relying on these examples, we present what is needed to write a fully abstract compiler from $\mathsf{L^U}$ to $\mathbf{L^P}$, the languages of our first compiler (Section 6.3). We use this compiler to compare *RSC* and *FAC* concretely, showing that, at least on this example, *RSC* permits more efficient code and affords simpler proofs than *FAC* (Section 6.4).

Remark. This does not imply that one should always prefer *RSC* to *FAC* blindly. In some cases, one may want to establish full abstraction for reasons other than security, so there *FAC* is preferable. Also, when the target language is typed [11, 12, 23, 54] or has abstractions similar to those of the source, full abstraction may have no downsides (in terms of efficiency of compiled code and simplicity of proofs) relative to *RSC*. However, in many settings, including those we consider, target languages are not typed, and often differ significantly from the source in their abstractions. In such cases, *RSC* is a worthy alternative.

6.1 Formalising Fully Abstract Compilation

As stated in Section 1, *FAC* requires the preservation and reflection of observational equivalence, and most existing work instantiates observational equivalence with contextual equivalence ($\simeq_{ctx}$). Contextual equivalence and *FAC* are defined below. Informally, two components C_1 and C_2 are contextually equivalent if no context A interacting with them can tell them apart, i.e., they are *indistinguishable*. Contextual equivalence can encode security properties such as confidentiality, integrity, invariant maintenance and non-interference [6, 10, 59, 64]. We do not explain this well-known observation here, but refer the interested reader to the survey of Patrignani *et al.* [60]. Informally, a compiler $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ is fully abstract if it translates (only) contextually-equivalent source components into contextually-equivalent target ones.

Definition 6.1 (Contextual equivalence and fully abstract compilation).

$$C_1 \simeq_{ctx} C_2 \overset{\text{def}}{=} \forall A.\, A\,[C_1] \Uparrow \iff A\,[C_2] \Uparrow, \text{ where } \Uparrow \text{ means execution divergence}$$

$$\vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : FAC \overset{\text{def}}{=} \forall \mathsf{C_1}, \mathsf{C_2}.\, \mathsf{C_1} \simeq_{ctx} \mathsf{C_2} \iff [\![\mathsf{C_1}]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{C_2}]\!]^{\mathsf{S}}_{\mathbf{T}}$$

The security-relevant direction of *FAC* is $\Rightarrow$ [31]. This direction is security-relevant because the proof thesis concerns target contextual equivalence ($\simeq_{ctx}$). Unfolding the definition of $\simeq_{ctx}$ on the right of the implication yields a universal quantification over all possible target contexts $\mathbf{A}$, which captures malicious attackers. In fact, there may be target contexts $\mathbf{A}$ that can interact with compiled code in ways that are impossible in the source language. Compilers that attain *FAC* with untyped target languages often insert checks in compiled code that detect such interactions and respond to them securely [64], often by halting the execution [6, 10, 31, 40, 42, 45, 59, 60]. These checks are often inefficient, but must be performed even if the interactions are not security-relevant. We now present examples of this.

6.2 *FAC* and Inefficient Compiled Code

We illustrate various ways in which *FAC* forces inefficiencies in compiled code via a running example. Consider a password manager written in an object-oriented language that is compiled to an assembly-like language. We elide most code details and focus only on the relevant aspects.

```
private db: Database;

public testPwd( user: Char[8], pwd: BitString): Bool{
  if( db.contains( user )){ return db.get( user ).getPassword() == pwd; }
}
...
private class Database{ ... }
```

The source program exports the function `testPwd` to check whether a `user`'s stored password matches a given password `pwd`. The stored password is in a local database, which is represented by a piece of *local state* in the variable `db`. The details of `db` are not important here, but the database is marked private, so it is not directly accessible to the context of this program in the source language.

Example 6.2 (Extensive checks). A fully-abstract compiler for the program above must generate code that checks that the arguments passed to `testPwd` by the context are of the right type [10, 35, 42, 59, 63]. The code expects an array of characters of length 8. A parameter of a different type (e.g., an array of objects) cannot be passed in the source, so it must also be prevented in the target. Since the target is untyped, code must be inserted to check the argument. Specifically, a fully abstract compiler will generate code similar to the following (we assume that arrays are passed as pointers into the heap).

```
label testpwd
  for i = 0; i< 8; i++  // 8 is the legth of the user field in the previous snippet
    load the memory word stored at address r0+ i into r1
    test that r1 is a valid char encoding
  ...
```

Basically, this code dynamically checks that the first argument is a character array of length 8 because a type mismatch could lead to a violation of *FAC*. Such a check can be very inefficient when the length is very long. ⊡

The problem here is that *FAC* forces these checks on all arguments, even those that have no security relevance. In contrast, *RSC* does not need these checks. Indeed, none of our earlier compilers ($\llbracket\cdot\rrbracket^{\mathbf{L^U}}_{\mathbf{L^P}}$, $\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$ and $\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{L^I}$) insert them. Note that any robustly safe source program will already have programmer-inserted checks for all parameters that are relevant to the safety property of interest, and these checks will be compiled to the target. For other parameters, the checks are irrelevant, both in the source and the target, so there is no need to insert them.

Example 6.3 (Component size in memory). Let us now consider two different ways to implement the `Database` class: as a `List` and as a `RedBlackTree`. As the class is `private`, its internal behaviour and representation of the database is invisible to the outside. Let $\mathsf{C}_{\mathsf{list}}$ be the program with the `List` implementation and $\mathsf{C}_{\mathsf{tree}}$ be the program with the `RedBlackTree` implementation; in the source language, these are equivalent.

However, a subtlety arises when considering the assembly-level, compiled counterparts of $\mathsf{C}_{\mathsf{list}}$ and $\mathsf{C}_{\mathsf{tree}}$: the *code* of a `RedBlackTree` implementation consumes more memory than the code of a `List` implementation. Thus, a target-level context can distinguish $\mathsf{C}_{\mathsf{list}}$ from $\mathsf{C}_{\mathsf{tree}}$ by just inspecting the sizes of the code segments. So, in order for the compiler to be fully abstract, it must produce code of a fixed size [10, 59]. This wastes memory and makes it impossible to compile some components. An alternative would be to spread the components in an overly-large memory at random places i.e., use address-space layout randomization or ASLR, so that detecting different code sizes has a negligible chance of success [6, 40]. However, ASLR is now known to be broken [16, 41]. ⊡

Again, we see that *FAC* introduces an inefficiency in compiled code (pointless code memory consumption) even though this has no security implication here. In contrast, *RSC* does not require this unless the safety property(ies) of interest care about the size of the code (which is very unlikely in a security context, since security by code obscurity is a strongly discouraged practice).

Example 6.4 (Wrappers for heap resources). Assume that the `Database` class is implemented as a `List`. Shown below are two implementations of the `newList` method inside `List` which we call $\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$. The only difference between $\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$ is that $\mathsf{C}_{\mathsf{two}}$ allocates two lists internally; one of these (`shadow`) is used for internal purposes only.

```
public newList(): List{

  ell = new List();
  return ell;
}
```

```
public newList(): List{
  shadow = new List();
  ell = new List();
  return ell;
}
```

Again, C_{one} and C_{two} are equivalent in a source language that does not allow pointer comparison. To attain *FAC* when the target allows pointer comparisons, the pointers returned by `newList` in the two implementations must be the same, but this is very difficult to ensure since the second implementation does more allocations. A simple solution to this problem is to wrap `ell` in a proxy object and return the proxy [10, 50, 59, 63]. Compiled code needs to maintain a lookup table mapping the proxy to the original object. Proxies must have allocation-independent addresses. Proxies work but they are inefficient due to the need to look up the table on every object access.

Another way to attain *FAC* is to weaken the source language, introducing an operation to distinguish object identities in the source [56]. However, this is a widely discouraged practice, as it changes the source language from what it really is and the implication of such a change may be difficult to fathom for programmers and verifiers alike. ⊡

In this example, *FAC* forces all privately allocated locations to be wrapped in proxies, but *RSC* does not require this. Our target languages $\mathbf{L^P}$, $\mathbf{L}^{\pi}$ and L^I support address comparison (addresses are natural numbers or integers in their heaps) but $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ and $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ just use capabilities to attain security efficiently, while $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^I}$ just relies on enclaves. On the other hand, for attaining *FAC*, capabilities or enclaves would be insufficient since they do not hide addresses; proxies would still be required (this point is concretely demonstrated in Section 6.3).

Example 6.5 (Strict termination vs divergence). Consider a source language that is strictly terminating while a target language that is not. Below is an extension of the password manager to allow database encryption via an externally-defined function. As the database is not directly accessible from external code, the two implementations below C_{enc} (which does the encryption) and C_{skip} which skips the encryption are equivalent in the source.

```
public encryptDB( func : Database →
     Bitstring) : void {
  func( this.db );
  return;
}
```

```
public encryptDB( func : Database →
     Bitstring) : void {

  return;
}
```

If we compile C_{enc} and C_{skip} to an assembly language, the compiled counterparts *cannot* be equivalent, since the target-level context can detect which function is compiled by passing a `func` that diverges. Calling the compilation of C_{enc} with such a `func` will cause divergence, while calling the compilation of C_{skip} will immediately return. ⊡

This case presents a situation where *FAC* is outright *impossible*. The only way to get *FAC* is to make the source language artificially non-terminating. (See the work of Devriese *et al.* [32] for more details of this particular problem.) On the other hand, *RSC* can be easily attained even in such settings since it is completely independent of termination in the languages (unless the safety properties of interest are termination-sensitive, which is usually not the case). For the specific examples we have considered, even if our source languages $\mathsf{L^U}$ and L^{τ} were restricted to terminating programs only, the same compilers and the same proofs of *RSC* would still work.

Remark. It is worth noting that many of the inefficiencies above might be resolved by just replacing contextual equivalence with a different equivalence in the statement of *FAC*. However, it is not known how to do this generally for arbitrary sources of inefficiency and, further, it is unclear what the security consequences of such instantiations of *FAC* would be. On the other hand, *RSC* is *uniform* and it does address all these inefficiencies.

An issue that can normally not be addressed just by tweaking equivalences is side-channel leaks, as they are, by definition, not expressible in the language. Neither *FAC* nor *RSC* deals with side channels, but recent results describe how to account for side channels in security-preserving compilers [17].

6.3 Towards a Fully Abstract Compiler from L^{U} to $\mathbf{L}^{\mathbf{P}}$

In this section, we describe what it would take to build a fully abstract compiler from L^{U} to $\mathbf{L}^{\mathbf{P}}$. Along the way, we note how this compiler would be less efficient than the *RSC* compiler we described earlier. In fact, to get a fully abstract compiler, we need to adjust the languages. We describe these language changes first.

6.3.1 Language Extensions to L^{U} and $\mathbf{L}^{\mathbf{P}}$. This section lists the language extensions required for a fully-abstract compiler from L^{U} to $\mathbf{L}^{\mathbf{P}}$. It is not possible to motivate all the language changes before explaining the details of the compiler, so some of the justification is postponed to Section 6.3.2.

A first concern for full abstraction is that a target context can always determine the memory consumption of two compiled components, analogously to Example 6.3. To ensure that this does not break full abstraction, we add a source expression size that returns the number of locations ℓ allocated in the heap.

In the target language $\mathbf{L}^{\mathbf{P}}$, we need to know whether an expression is a pair, whether it is a location, and we need to be able to compare two capabilities. Accordingly, we add the operations $\mathbf{isloc(e)}$, $\mathbf{ispair(e)}$ and $\mathbf{eqcap(e, e)}$, respectively.

Finally, compiled code needs private functions for its runtime checks that must not be visible to the context. $\mathbf{L}^{\mathbf{P}}$ does not have this functionality: all functions defined by a component can be called by the context. Accordingly, we modify $\mathbf{L}^{\mathbf{P}}$ so that all functions $\overline{\mathbf{F}}$ defined in a component are private to it by default. Each component explicitly includes the list of functions it exports; only these functions can be called by the context.

6.3.2 The $\wr\cdot\int_{\mathbf{L}^{\mathbf{P}}}^{\mathsf{L}^{\mathsf{U}}}$ Compiler. The fully abstract compiler $\wr\cdot\int_{\mathbf{L}^{\mathbf{P}}}^{\mathsf{L}^{\mathsf{U}}}$ is similar to the *RSC* attaining compiler $[\![\cdot]\!]_{\mathbf{L}^{\mathbf{P}}}^{\mathsf{L}^{\mathsf{U}}}$, but with critical differences. We know that fully abstract compilation preserves all source abstractions in the target language. Here, the only abstraction that distinguishes $\mathbf{L}^{\mathbf{P}}$ from L^{U} is that locations are abstract in L^{U}, but concrete natural numbers in $\mathbf{L}^{\mathbf{P}}$. Thus, locations allocated by compiled code must not be passed directly to the context as this would reveal the allocation order (as seen in Example 6.4). Instead of passing the location $\langle\mathbf{n}, \mathbf{k}\rangle$ to the context, the compiler arranges for an opaque handle $\langle\mathbf{n}', \mathbf{k}_{\mathbf{com}}\rangle$ (that cannot be used to access any location directly) to be passed. Such an opaque handle is often called a *mask* or *seal* in the literature and this technique is often called dynamic sealing [74].

To ensure that masking is done properly, $\wr\cdot\int_{\mathbf{L}^{\mathbf{P}}}^{\mathsf{L}^{\mathsf{U}}}$ inserts code at entry points and at exit points to compiled code (i.e., at function calls, before returning, before and after callbacks), *wrapping* the compiled code in a way that enforces masking. This notion of wrapping is standard in literature on fully abstract compilation [35, 63]. The wrapper keeps a list $\overline{\mathbf{L}}$ of component-allocated locations that are shared with the context in order to know their masks. When a component-allocated location is shared, it is added to the list $\overline{\mathbf{L}}$. The mask of a location is its index in this list. If the same location is shared again it is not added again but its previous index is used. So if $\langle\mathbf{n}, \mathbf{k}\rangle$ is the 4th element of $\overline{\mathbf{L}}$, its mask is $\langle\mathbf{4}, \mathbf{k}_{\mathbf{com}}\rangle$. To implement lookup in $\overline{\mathbf{L}}$ we must compare capabilities too, for we rely on the newly added operation $\mathbf{eqcap}$. To ensure capabilities do not leak to the context, the second field of the pair is a constant capability $\mathbf{k}_{\mathbf{com}}$, which protects a dummy location the compiled code does not actually use. Technically speaking, this is exactly how existing fully abstract compilers operate (e.g., as in the work of Patrignani *et al.* [59]).

As should be clear, this kind of masking is very inefficient at runtime. However, even this masking is not sufficient for full abstraction. Next, we explain additional things the compiler must do.

Determining when a Location is Passed to the Context. A component-allocated location can be passed to the context not just as a function argument but on the heap. So before passing control to the context the compiled code needs to scan the whole heap where a location can be passed and mask any component-allocated locations it finds. Dually, when receiving control the compiled code must scan the heap to unmask all masked locations. The problem now is determining what parts of the heap to scan and how. Specifically, the compiled code needs to keep track of all the locations (and related capabilities) that are shared, i.e., (i) passed from the context to the component and (ii) passed from the component to the context. These are the locations through which possible communication of locations can happen. Compiled code keeps track of these shared locations in a list $\overline{S}$. Intuitively, on the first function call from the context to the compiled component, assuming the parameter is a location, the compiled code will register that location and all other locations reachable from it in $\overline{S}$. On subsequent ? (incoming) actions, the compiled code will register all new locations available as parameters or reachable from $\overline{S}$. Then, on any ! (outgoing) action, the compiled code must scan whatever locations (that the compiled code has created) are now reachable from $\overline{S}$ and add them to $\overline{S}$. We need the new instructions **isloc** and **ispair** in L^P to compute these reachable locations. Of course, this kind of scanning of locations reachable from $\overline{S}$ at every call/return between components can be extremely costly.

Enforcing the Masking of Locations. The functions **mask** and **unmask** are added by the compiler to the compiled code. The first function takes a location (which intuitively contains a value **v**) and replaces (in **v**) any pair $\langle n, k\rangle$ of a location protected with a component-created capability **k** with its index in the masking list $\overline{L}$. The second function replaces any pair $\langle n, k_{com}\rangle$ with the *n*th element of the masking list $\overline{L}$. These functions should not be directly accessible to the context (else it can **unmask** any **mask**ed location and break full abstraction). This is why L^P needs private functions.

Letting the Context use Masked Locations. Masked locations cannot be used directly by the context for reading and writing. Thus, compiled code must provide a **read** and a **write** function (both of which are public) that implement reading and writing for masked locations.

As should be clear, code compiled through $\wr\cdot\int^{L^U}_{L^P}$ has a lot of runtime overhead in calculating the heap reachable from $\overline{S}$ and in **mask**ing and **unmask**ing locations. Additionally, it also has code memory overhead: the functions **read**, **write**, **mask**, **unmask** and list manipulation code must be included. Finally, there is data overhead in maintaining $\overline{S}$, $\overline{L}$ and other supporting data structures to implement the runtime checks described above. In contrast, the code compiled through $[\![\cdot]\!]^{L^U}_{L^P}$ (which is just robustly safe and not fully abstract) has none of these overheads.

6.4 Proving that $\wr\cdot\int^{L^U}_{L^P}$ is a Fully Abstract Compiler

Using $\wr\cdot\int^{L^U}_{L^P}$ as a concrete example, we now discuss why *proving FAC* can be harder than proving *RSC*. Consider the hard part of *FAC*, the forward implication, $C_1 \simeq_{ctx} C_2 \Rightarrow [\![C_1]\!]^S_T \simeq_{ctx} [\![C_2]\!]^S_T$. The contrapositive of this statement is $[\![C_1]\!]^S_T \not\simeq_{ctx} [\![C_2]\!]^S_T \Rightarrow C_1 \not\simeq_{ctx} C_2$. By unfolding the definition of $\not\simeq_{ctx}$ we see that, given a target context A that distinguishes $[\![C_1]\!]^S_T$ from $[\![C_2]\!]^S_T$, it is necessary to show that there exists a source context A that distinguishes C_1 from C_2. That source context A

must be built (backtranslated) starting from the already given target context $\mathbf{A}$ that differentiates $[\![\mathsf{C}_1]\!]^{\mathsf{S}}_{\mathbf{T}}$ from $[\![\mathsf{C}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$.

A backtranslation directed by the syntax of the target context $\mathbf{A}$ is hopeless here since the target expressions **iscap** and **isloc** cannot be directly backtranslated to valid source expressions. Hence, we resort to another well-known technique [10, 63]. First, we define a *fully abstract (labeled) trace semantics* for the target language. A trace semantics is fully abstract when two components are contextually inequivalent iff their trace semantics differ in at least one trace. So if we write $\mathsf{TR}\left(\mathbf{C}\right)$ to denote the traces of the component $\mathbf{C}$, we can formally state full abstraction of the trace semantics as: $\mathsf{TR}\left(\wr \mathsf{C}_1 \int^{\mathsf{L^U}}_{\mathbf{L^P}}\right) = \mathsf{TR}\left(\wr \mathsf{C}_2 \int^{\mathsf{L^U}}_{\mathbf{L^P}}\right) \iff \wr \mathsf{C}_1 \int^{\mathsf{L^U}}_{\mathbf{L^P}} \simeq_{ctx} \wr \mathsf{C}_2 \int^{\mathsf{L^U}}_{\mathbf{L^P}}$. Given this trace semantics, the statement of the forward implication of full abstraction reduces to:

$$\mathsf{TR}\left(\wr \mathsf{C}_1 \int^{\mathsf{L^U}}_{\mathbf{L^P}}\right) \neq \mathsf{TR}\left(\wr \mathsf{C}_2 \int^{\mathsf{L^U}}_{\mathbf{L^P}}\right) \Rightarrow \mathsf{C}_1 \not\simeq_{ctx} \mathsf{C}_2.$$

The advantage of this formulation over the original one is that now we can construct a distinguishing source context for C_1 and C_2 using the *trace* on which $\mathsf{TR}\left(\wr \mathsf{C}_1 \int^{\mathsf{L^U}}_{\mathbf{L^P}}\right)$ and $\mathsf{TR}\left(\wr \mathsf{C}_2 \int^{\mathsf{L^U}}_{\mathbf{L^P}}\right)$ disagree. While this proof strategy of constructing a source context from a trace is similar to our proof of *RSC*, it is fundamentally much harder and much more involved. There are two reasons for this.

First, fully abstract trace semantics are much more complex than our simple trace semantics of $\mathbf{L^P}$ from earlier sections. The reason is that our earlier trace semantics include the entire heap in every action, but this breaks full abstraction of the trace semantics: such trace semantics also distinguish contextually equivalent components that differ in their internal private state. In a fully abstract trace semantics, the trace actions must record *only* those heap locations that are shared between the component and the context. Consequently, the definition of the trace semantics must inductively track what has been shared in the past. In particular, the definition must account for locations reachable indirectly from explicitly shared locations. This complicates both the definition of traces and the proofs that build on the definition.

Second, the source context that the backtranslation constructs from a target trace must simulate the shared part of the heap at every context switch. Since locations in the target may be masked now, the source context must maintain a map with the source locations corresponding to the target masked ones, which complicates the source context substantially. Call this map B. Now, this affects two patterns of target traces that need to be handled in a special way: `call` **read v H?** · `ret` **H′!** and `call` **write v H?** · `ret` **H′!**. Normally, these patterns would be translated to source-level calls to the same functions (read and write), but this is not possible. In fact, the source code has no read or write function, and the target-level calls to these functions need to be backtranslated to the corresponding source constructs (! and :=, respectively). The locations used by these constructs must be looked up from B as these are reads and writes to masked locations. Moreover, calls and returns to **read** can be simply ignored since the effects of reads are already captured by later actions in traces. Calls and returns to **write** cannot be ignored as they set up a component location (albeit masked) in a certain way and that affects the behaviour of the component. We show in Example 6.6 how to backtranslate calls and returns to **write**.

Example 6.6 (Backtranslation of traces). Consider the trace below and its backtranslation.

$$\begin{array}{ll} (1) & \mathtt{call}\ \mathbf{f\ 0\ 1} \mapsto \mathbf{4?} \\ (2) & \mathtt{ret}\ \mathbf{1} \mapsto \langle \mathbf{1}, \mathbf{k_{com}} \rangle\ \mathbf{!} \\ (3) & \left[\begin{array}{l} \mathtt{call}\ \mathbf{write}\ \langle \langle \mathbf{1}, \mathbf{k_{com}} \rangle, \mathbf{5} \rangle\ \mathbf{1} \mapsto \langle \mathbf{1}, \mathbf{k_{com}} \rangle\ \mathbf{?} \\ \mathtt{ret}\ \mathbf{1} \mapsto \langle \mathbf{1}, \mathbf{k_{com}} \rangle\ \mathbf{!} \end{array}\right. \end{array}
\qquad
\begin{array}{l} \mathsf{main(x)} \mapsto \\ \quad \left.\begin{array}{l} \mathsf{let\ x = new\ 4\ in\ L :: \langle x, 1 \rangle} \\ \mathsf{call\ f\ 0} \end{array}\right] (1) \\ \quad \mathsf{let\ x = !L(1)\ in\ B :: \langle x, 1 \rangle} \quad]\ (2) \\ \quad \mathsf{!B(1) := 5} \quad]\ (3) \end{array}$$

The first action, where the context registers the first location in the list L, is as before. Then in the second action the compiled component passes to the context (in location 1) a masked location with index 1 and, later, the context writes 5 to it. The backtranslated code must recognise this pattern and store the location that, in the source, corresponds to the mask 1 in the list B (action 2). In action 3, when it is time to write 5 to that location, the code looks up the location to write to from B. ⊡

It should be clear that this proof of *FAC* is substantially harder than our corresponding proof of *RSC*, which needed neither fully abstract traces, nor tracking any mapping in the backtranslated source contexts.

7 RELATED WORK

Recent work [9, 36] presents new criteria for secure compilation that ensure preservation of subclasses of hyperproperties. Hyperproperties [27] are a formal representation of predicates on programs, i.e., they are predicates on sets of traces. Hyperproperties capture many security-relevant properties including not just conventional safety and liveness, which are predicates on traces, but also properties like non-interference, which is a predicate on pairs of traces. Modulo technical differences, our definition of *RSC* coincides with the criterion of "robust safety property preservation" in [9, 36]. We show, through concrete instances, that this criterion can be easily realized by compilers, and develop two proof techniques for establishing it. We further show that the criterion leads to more efficient compiled code than does *FAC*. Additionally, the criteria in [9, 36] assume that behaviours in the source and target are represented using the same alphabet. Hence, the definitions (somewhat unrealistically or ideally) do not require a translation of source properties to target properties. That line of work has been extended to consider criteria that preserve hyperproperties between languages with different trace models which are connected by a trace relation similar to ours [8]. Like this last work, we consider differences in the representation of behaviour in the source and in the target and this is accounted for in our monitor relation $\mathsf{M} \approx \mathsf{M}$. Unlike this last work, we provide different instances where the relation is instantiated in order to show how the theory scales to different protection mechanisms. A slightly different account of the difference between traces across languages is presented by Patrignani and Garg [64] in the context of reactive black-box programs.

Abate *et al.* [7] define a variant of robustly-safe compilation called RSCC specifically tailored to the case where (source) components can perform undefined behaviour. RSCC does not consider attacks from arbitrary target contexts but from compiled components that can become compromised and behave in arbitrary ways. To demonstrate RSCC, Abate *et al.* [7] rely on two backends for their compiler: software fault isolation and tag-based monitors. On the other hand, we rely on capability machines and memory isolation. RSCC also preserves (a form of) safety properties and can be achieved by relying on a trace-based backtranslation; it is unclear whether proofs can be simplified when the source is verified and concurrent, as in our second compiler.

ASLR [6, 40], protected module architectures [10, 45, 59, 63], tagged architectures [42], capability machines [78] and cryptographic primitives [4, 5, 24, 28] have been used as targets for *FAC*. We believe all of these can also be used as targets of *RSC*-attaining compilers. In fact, some targets such as capability machines seem to be better suited to *RSC* than *FAC*, as we demonstrated.

Ahmed *et al.* prove full abstraction for several compilers between typed languages [11, 12, 54]. As compiler intermediate languages are often typed, and as these types often serve as the basis for complex static analyses, full abstraction seems like a reasonable goal for (fully typed) intermediate compilation steps. In the last few steps of compilation, where the target languages are unlikely to be typed, one could establish robust safety preservation and combine the two properties (vertically) to get an end-to-end security guarantee.

There are three other criteria for secure compilation that we would like to mention: securely compartmentalised compilation (SCC) [42], trace-preserving compilation (TPC) [64] and non-interference-preserving compilation (NIPC) [13, 17, 18, 29, 51]. SCC is a re-statement of the "hard" part of full abstraction (the forward implication), but adapted to languages with undefined behaviour and a static notion of components. Thus, SCC suffers from much of the same efficiency drawbacks as *FAC*. TPC is a stronger criterion than *FAC*, that most existing fully abstract compilers also attain. Again, compilers attaining TPC also suffer from the drawbacks of compilers attaining *FAC*.

NIPC preserves a single property: noninterference (NI). However, this line of work does not consider active target-level adversaries yet. Instead, the focus is on compiling whole programs. Since noninterference is not a safety property, it is difficult to compare NIPC to *RSC* directly. However, noninterference can also be approximated as a safety property [22]. So, in principle, *RSC* (with adequate massaging of observations) can be applied to stronger end-goals than NIPC.

Swamy *et al.* [75] embed an F^* model of a gradually and robustly typed variant of JavaScript into an F^* model of JavaScript. Gradual typing supports constructs similar to our endorsement construct in L^{τ}. Their type-directed compiler is proven to attain memory isolation as well as static and dynamic memory safety. However, they do not consider general safety properties, nor a general criterion for compiler security.

Two of our target languages rely on capabilities for restricting access to sensitive locations from the context. Although capabilities are not mainstream in any processor, fully functional research prototypes such as Cheri exist [81]. Capability machines have previously been advocated as a target for efficient secure compilation [33] and preliminary work on compiling C-like languages to them exists, but the criterion applied is *FAC* [70, 71, 78].

On the other hand, one of our target languages relies on coase-grained isolation, a feature that is being increasingly supported in hardware (Intel calls this SGX [49]; ARM calls it TrustZone [77]). Coarse-grained isolation has also been advocated as a target for secure compilation [10, 45, 57]. The criterion applied in these works is *FAC*, which is what lets us draw a starker comparison in Section 6.

8 CONCLUSION

This paper has examined robustly safe compilation (*RSC*), a soundness criterion for compilers with direct relevance to security. We have shown that the criterion is easily realizable and may lead to more efficient code than does fully abstract compilation. We have also presented two techniques for establishing that a compiler attains *RSC*. One is an adaptation of an existing technique, backtranslation, and the other is based on inductive invariants.

ACKNOWLEDGMENTS

The authors would like to thank Dominique Devriese, Akram El-Korashy, Cătălin Hriţcu, Frank Piessens, David Swasey and the anonymous reviewers of ESOP'19 and of TOPLAS for useful feedback and discussions on an earlier draft.

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762) and by the Office of Naval Research for support through grant N00014-18-1-2620, Accountable Protocol Customization.

REFERENCES

[1] Martín Abadi. 1999. Protection in programming-language translations. In *Secure Internet programming*. Springer-Verlag, London, UK, 19–34. http://dl.acm.org/citation.cfm?id=380171.380174

[2] Martín Abadi. 1999. Secrecy by Typing in Security Protocols. *J. ACM* 46, 5 (Sept. 1999), 749–786. DOI:http://dx.doi.org/10.1145/324133.324266

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (2009), 40 pages. DOI:http://dx.doi.org/10.1145/1609956.1609960

[4] Martín Abadi, Cédric Fournet, and Georges Gonthier. 2000. Authentication Primitives and their Compilation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. ACM, New York, NY, USA, 302–315. DOI:http://dx.doi.org/10.1145/325694.325734

[5] Martín Abadi, Cédric Fournet, and Georges Gonthier. 2002. Secure Implementation of Channel Abstractions. *Information and Computation* 174 (2002), 37–83. DOI:http://dx.doi.org/10.1006/inco.2002.3086

[6] Martín Abadi and Gordon D. Plotkin. 2012. On Protection by Layout Randomization. *ACM Transactions on Information and System Security* 15, Article 8 (July 2012), 29 pages. DOI:http://dx.doi.org/10.1145/2240276.2240279

[7] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1351–1368. DOI:http://dx.doi.org/10.1145/3243734.3243745

[8] Carmine Abate, Roberto Blanco, Stefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hritcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 1–28.

[9] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *2019 IEEE 32th Computer Security Foundations Symposium (CSF 2019)*. DOI:http://dx.doi.org/10.1109/CSF.2019.00025

[10] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. 2012. Secure Compilation to Modern Processors. In *2012 IEEE 25th Computer Security Foundations Symposium (CSF 2012)*. IEEE, 171–185. DOI:http://dx.doi.org/10.1109/CSF.2012.12

[11] Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 157–168. DOI:http://dx.doi.org/10.1145/1411204.1411227

[12] Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 431–444. DOI:http://dx.doi.org/10.1145/2034773.2034830

[13] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *ACM Conference on Computer and Communications Security*. ACM, 1807–1823.

[14] Bowen Alpern and Fred B. Schneider. 1985. Defining Liveness. *Inf. Process. Lett.* 21, 4 (1985), 181–185. DOI:http://dx.doi.org/10.1016/0020-0190(85)90056-0

[15] Michael Backes, Catalin Hritcu, and Matteo Maffei. 2014. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security* 22, 2 (2014), 301–353. DOI:http://dx.doi.org/10.3233/JCS-130493

[16] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. 2015. CAIN: Silently Breaking ASLR in the Cloud. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/woot15/workshop-program/presentation/barresi

[17] G. Barthe, B. Grégoire, and V. Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 328–343. DOI:http://dx.doi.org/10.1109/CSF.2018.00031

[18] Gilles Barthe, Tamara Rezk, and Amitabh Basu. 2007. Security Types Preserving Compilation. *Computer Languages, Systems and Structures* 33 (2007), 35–59. DOI:http://dx.doi.org/10.1016/j.cl.2005.05.002

[19] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (Feb. 2011), 45 pages. DOI:http://dx.doi.org/10.1145/1890028.1890031

[20] Nick Benton and Chung-kil Hur. 2010. *Realizability and Compositional Compiler Correctness for a Polymorphic Language.* Technical Report. MSR.

[21] Gérard Berry and Gérard Boudol. 1992. The Chemical Abstract Machine. *Theor. Comput. Sci.* 96, 1 (1992), 217–248.

[22] Gérard Boudol. 2009. Secure Information Flow As a Safety Property. Springer-Verlag, Berlin, Heidelberg, Chapter Formal Aspects in Security and Trust, 20–34. DOI:http://dx.doi.org/10.1007/978-3-642-01465-9_2

[23] William J. Bowman and Amal Ahmed. 2015. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM, New York, NY, USA.

[24] Michele Bugliesi and Marco Giunti. 2007. Secure Implementations of Typed Channel Abstractions. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 251–262. DOI:http://dx.doi.org/10.1145/1190216.1190253

[25] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-based Addressing. *SIGPLAN Not.* 29 (1994), 319–327. DOI:http://dx.doi.org/10.1145/195470.195579

[26] Stephen Chong. 2008. *Expressive and Enforceable Information Security Policies.* Ph.D. Dissertation. Cornell University.

[27] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (Sept. 2010), 1157–1210.

[28] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James Leifer. 2008. A Secure Compiler for Session Abstractions. *Journal of Computer Security* 16 (2008), 573–636. http://dl.acm.org/citation.cfm?id=1454415.1454419

[29] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-End Verification of Information-Flow Security for C and Assembly Programs. *SIGPLAN Not.* 51, 6 (June 2016), 648–664. DOI:http://dx.doi.org/10.1145/2980983.2908100

[30] Dominique Devriese, Marco Patrignani, Steven Keuchel, and Frank Piessens. 2017. Modular, Fully-Abstract Compilation by Approximate Back-Translation. *Logical Methods in Computer Science* Volume 13, Issue 4 (Oct. 2017).

[31] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-Abstract Compilation by Approximate Back-Translation. *SIGPLAN Not.* 51, 1 (Jan. 2016), 164–177. DOI:http://dx.doi.org/10.1145/2914770.2837618

[32] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2017. Parametricity versus the Universal Type. *Proc. ACM Program. Lang.* 2, POPL, Article 38 (Dec. 2017), 23 pages. DOI:http://dx.doi.org/10.1145/3158126

[33] Akram El-Korashy. 2016. *A Formal Model for Capability Machines – An Illustrative Case Study towards Secure Compilation to CHERI.* Master's thesis. Universitat des Saarlandes.

[34] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2007. A Type Discipline for Authorization Policies. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 25 (Aug. 2007). DOI:http://dx.doi.org/10.1145/1275497.1275500

[35] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully Abstract Compilation to JavaScript. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 371–384. DOI:http://dx.doi.org/10.1145/2429069.2429114

[36] D. Garg, C. Hritcu, M. Patrignani, M. Stronati, and D. Swasey. 2017. Robust Hyperproperty Preservation for Secure Compilation (Extended Abstract). *ArXiv e-prints* (Oct. 2017). arXiv:cs.CR/1710.07309

[37] Andrew D. Gordon and Alan Jeffrey. 2003. Authenticity by Typing for Security Protocols. *J. Comput. Secur.* 11, 4 (July 2003), 451–519. http://dl.acm.org/citation.cfm?id=959088.959090

[38] Daniele Gorla and Uwe Nestmann. 2016. Full abstraction for expressiveness: history, myths and facts. *Mathematical Structures in Computer Science* 26, 4 (2016), 639–654. DOI:http://dx.doi.org/10.1017/S0960129514000279

[39] Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation between ML and Assembly. *SIGPLAN Not.* 46, 1 (Jan. 2011), 133–146. DOI:http://dx.doi.org/10.1145/1925844.1926402

[40] Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. 2011. Local Memory via Layout Randomization. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium (CSF '11)*. IEEE Computer Society, Washington, DC, USA, 161–174. DOI:http://dx.doi.org/10.1109/CSF.2011.18

[41] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 380–392. DOI:http://dx.doi.org/10.1145/2976749.2978321

[42] Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In *29th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press. http://arxiv.org/abs/1602.04503 To appear.

[43] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 178–190. DOI:http://dx.doi.org/10.1145/2837614.2837642

[44] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 147–163. http://dl.acm.org/citation.cfm?id=2685048.2685061

[45] Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. 2015. A Secure Compiler for ML Modules. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*. 29–48.

[46] Xavier Leroy. 2006. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. *SIGPLAN Not.* 41, 1 (Jan. 2006), 42–54. DOI:http://dx.doi.org/10.1145/1111320.1111042

[47] Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. DOI: http://dx.doi.org/10.1007/s10817-009-9155-4

[48] Sergio Maffeis, Martín Abadi, Cédric Fournet, and Andrew D. Gordon. 2008. *Code-Carrying Authorization.* Springer Berlin Heidelberg, Berlin, Heidelberg, 563–579. DOI:http://dx.doi.org/10.1007/978-3-540-88313-5_36

[49] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP '13.* ACM, Article 10, 1 pages. DOI:http://dx.doi.org/10.1145/2487726.2488368

[50] James H. Morris, Jr. 1973. Protection in programming languages. *Commun. ACM* 16 (1973), 15–21. DOI:http://dx.doi.org/10.1145/361932.361937

[51] Kedar S. Namjoshi and Lucas M. Tabajara. 2020. Witnessing Secure Compilation. In *VMCAI 2020 (Lecture Notes in Computer Science)*, Vol. 11990. Springer, 1–22. DOI:http://dx.doi.org/10.1007/978-3-030-39322-9_1

[52] Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-Parametric Parametricity. *SIGPLAN Not.* 44, 9 (Aug. 2009), 135–148. DOI:http://dx.doi.org/10.1145/1631687.1596572

[53] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015).* ACM, 166–178.

[54] Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016).* ACM, New York, NY, USA, 103–116. DOI:http://dx.doi.org/10.1145/2951913.2951941

[55] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer-Verlag New York, USA.

[56] Joachim Parrow. 2016. General conditions for full abstraction. *Mathematical Structures in Computer Science* 26, 4 (2016), 655–657. DOI:http://dx.doi.org/10.1017/S0960129514000280

[57] Marco Patrignani. 2015. *The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures.* Ph.D. Dissertation. KU Leuven, Leuven, Belgium.

[58] Marco Patrignani. 2020. Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets. (2020). arXiv:cs.SE/2001.11334

[59] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, Article 6 (April 2015), 50 pages. DOI: http://dx.doi.org/10.1145/2699503

[60] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (Jan. 2019), 36 pages. DOI: http://dx.doi.org/10.1145/3280984

[61] Marco Patrignani, Dave Clarke, and Frank Piessens. 2013. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13) (LNCS)*, Vol. 8301. 176–191.

[62] Marco Patrignani, Dave Clarke, and Davide Sangiorgi. 2011. Ownership Types for the Join Calculus. In *FMOODS/FORTE 2011 (LNCS)*, Vol. 6722. 289–303.

[63] Marco Patrignani, Dominique Devriese, and Frank Piessens. 2016. On Modular and Fully Abstract Compilation. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF 2016).*

[64] Marco Patrignani and Deepak Garg. 2017. Secure Compilation and Hyperproperties Preservation. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium CSF 2017, Santa Barbara, USA (CSF 2017).*

[65] Marco Patrignani and Deepak Garg. 2018. Robustly Safe Compilation or, Efficient, Provably Secure Compilation. *CoRR* abs/1804.00489 (2018).

[66] Marco Patrignani and Deepak Garg. 2019. Robustly Safe Compilation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019 (ESOP'19).* https://arxiv.org/abs/1804.00489

[67] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and Principles. *J. Comput. Secur.* 17, 5 (Oct. 2009), 517–548. http://dl.acm.org/citation.cfm?id=1662658.1662659

[68] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The High-Level Benefits of Low-Level Sandboxing. In *47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2020).*

[69] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50. DOI:http://dx.doi.org/10.1145/353323.353382

[70] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities - Provably Safe Stack and Return Pointer Management. In *27th European Symposium on Programming, ESOP.* 475–501. DOI:http://dx.doi.org/10.1007/978-3-319-89884-1_17

[71] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *PACMPL* 3, POPL (2019), 19:1–19:28. https://dl.acm.org/citation.cfm?id=3290332

[72] Ian Stark. 1994. *Names and Higher-Order Functions.* Ph.D. Dissertation. University of Cambridge. DOI:http://dx.doi.org/10/cgnm Also available as Technical Report 363, University of Cambridge Computer Laboratory.

[73] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15).* ACM, New York, NY, USA, 275–287. DOI:http://dx.doi.org/10.1145/2676726.2676985

[74] Eijiro Sumii and Benjamin C. Pierce. 2004. A Bisimulation for Dynamic Sealing. *SIGPLAN Not.* 39, 1 (Jan. 2004), 161–172. DOI:http://dx.doi.org/10.1145/982962.964015

[75] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. *SIGPLAN Not.* 49, 1 (Jan. 2014), 425–437. DOI:http://dx.doi.org/10.1145/2578855.2535889

[76] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017. October 22 - 27, 2017.*

[77] ARM. ARMSecurity Technology. 2009. Building a Secure System using TrustZone Technology. ARM Technical White Paper. (2009).

[78] Stelios Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2017. Towards Automatic Compartmentalization of C Programs on Capability Machines. In *Workshop on Foundations of Computer Security 2017 August 21, 2017 (FCS 2017).*

[79] Claudio Vasconcelos and Antonio Ravara. 2016. The While language. (2016). arXiv:cs.PL/1603.08949

[80] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4 (1996), 167–187. Issue 2-3.

[81] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14).* IEEE Press, Piscataway, NJ, USA, 457–468. http://dl.acm.org/citation.cfm?id=2665671.2665740

[82] Stephan Arthur Zdancewic. 2002. *Programming Languages for Information Security.* Ph.D. Dissertation. Cornell University.